Support intergiciel pour la conception et le déploiement adaptatifs fiables, application aux bâtiments intelligents

Adja Ndeye Sylla

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Adja Ndeye SYLLA

Thèse dirigée par **Éric Rutten, Chargé de Recherche, INRIA Grenoble Rhône-Alpes**,

préparée au sein du **CEA et de l'INRIA Grenoble Rhône-Alpes** dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Support intergiciel pour la conception et le déploiement adaptatifs fiables, application aux bâtiments intelligents

Soutenance de thèse le **18/12/2017**
devant le jury composé de :

Monsieur Frank SINGHOFF
Professeur, Université de Brest, Examinateur, Président
Monsieur Lionel SENTURIER
Professeur, Université de Lille, Rapporteur
Madame Giovanna DI MARZO SERUGENDO
Professeur, Université de Genève, Rapporteur
Monsieur Maxime LOUVEL
Ingénieur de Recherche, CEA Grenoble, Examinateur, Encadrant
Monsieur Éric RUTTEN
Chargé de Recherche, INRIA Grenoble, Examinateur, Directeur de thèse
Madame Patricia STOLF
Maître de Conférences, Université Toulouse - Jean-Jaurès, Examinatrice

Remerciements

C'est avec un grand plaisir que j'écris cette section, non pas parce que c'est la plus facile à rédiger mais, parce que je tiens à exprimer toute ma reconnaissance

— aux rapporteurs, Madame Giovanna Di Marzo Serugendo et Monsieur Linonel Senturier, et à tous les membres du jury, pour avoir voulu évaluer mon travail ;
— au chef du LIALP, Monsieur Vincent Olive, pour m'avoir permis d'effectuer d'abord mon stage de fin d'études puis ma thèse au sein de son laboratoire ;
— à mon encadrant de stage, de début de thèse et membre de mon jury de soutenance, François Pacull, pour son aide, sa disponibilité et sa bonne humeur ;
— à mes encadrants de thèse, Maxime Louvel et Eric Rutten, pour leur disponibilité, leur aide, et leur soutien constant durant toute la thèse. Merci de m'avoir aider à m'améliorer aussi bien sur le plan scientifique que personnel ;
— à toutes les personnes qui m'ont soutenue sur le plan scientifique, Isa, Soguy, Gwenaël, pour avoir répondu à mes questions et/ou débuggé des programmes. Merci de m'avoir aidé à obtenir les résultats qui sont présentés par la suite ;
— à tous les membres du labo, tous les permanents et non permanents de fin 2014 à 2018, pour leur aide, leurs conseils et surtout la bonne ambiance qui m'a permis d'effectuer mon stage puis ma thèse dans de bonnes conditions ;
— à des personnes que j'apprécie beaucoup, tata Ndeye Coumba, El Hadji, Isa, Maha, Sanaa, Olivier, pour leurs qualités humaines. Merci pour votre aide (e.g., relecture de chapitres de la thèse, d'articles) et/ou vos encouragements ;
— à mes frères et sœurs, aussi bien de sang que de coeur, Cams, Baye, Mouhamed, Pape Samba, Mouhamadou, Diara, Geneviève, Roxana, Aida, Ndoumbé, Mously, pour tout ce que vous faites pour moi. Merci d'être toujours présents ;
— à mes parents, pour leur confiance et leur soutien sans faille dans toutes mes entreprises. Merci papa d'avoir fait la thèse avec moi (eh oui on était deux ! !) ;
— à toutes les personnes qui ont contribué à mes études de la primaire à la thèse, spécialement Ousmane Sow, pour m'avoir donné une formation de qualité.

Dieuredieuf yene gneupe. Amouma sene faye, Yallah rek laaa lène di bayel ! ! !

Résumé

Dans le contexte de l'informatique pervasive et de l'internet des objets, les systèmes sont hétérogènes, distribués et adaptatifs (p. ex., systèmes de gestion des transports, bâtiments intelligents). La conception et le déploiement de ces systèmes sont rendus difficiles par leur nature hétérogène et distribuée mais aussi le risque de décisions d'adaptation conflictuelles et d'inconsistances à l'exécution. Les inconsistances sont causées par des pannes matérielles ou des erreurs de communication. Elles surviennent lorsque des actions correspondant aux décisions d'adaptation sont supposées être effectuées alors qu'elles ne le sont pas.

Cette thèse propose un support intergiciel, appelé SICODAF, pour la conception et le déploiement de systèmes adaptatifs fiables. SICODAF combine une fiabilité comportementale (absence de décisions conflictuelles) au moyen de systèmes de transitions et une fiabilité d'exécution (absence d'inconsistances) à l'aide d'un intergiciel transactionnel. SICODAF est basé sur le calcul autonomique. Il permet de concevoir et de déployer un système adaptatif sous la forme d'une boucle autonomique qui est constituée d'une couche d'abstraction, d'un mécanisme d'exécution transactionnelle et d'un contrôleur. SICODAF supporte trois types de contrôleurs (basés sur des règles, sur la théorie du contrôle continu ou discret). Il permet également la reconfiguration d'une boucle, afin de gérer les changements d'objectifs qui surviennent dans le système considéré, et l'intégration d'un système de détection de pannes matérielles. Enfin, SICODAF permet la conception de boucles multiples pour des systèmes qui sont constitués de nombreuses entités ou qui requièrent des contrôleurs de types différents. Ces boucles peuvent être combinées en parallèle, coordonnées ou hiérarchiques.

SICODAF a été mis en œuvre à l'aide de l'intergiciel transactionnel LINC, de l'environnement d'abstraction PUTUTU et du langage Heptagon/BZR qui est basé sur des systèmes de transitions. SICODAF a été également évalué à l'aide de deux études de cas.

Abstract

In the context of pervasive computing and internet of things, systems are heterogeneous, distributed and adaptive (e.g., transport management systems, building automation). The design and the deployment of these systems are made difficult by their heterogeneous and distributed nature but also by the risk of conflicting adaptation decisions and inconsistencies at runtime. Inconsistencies are caused by hardware failures or communication errors. They occur when actions corresponding to the adaptation decisions are assumed to be performed but are not.

This thesis proposes a middleware support, called SICODAF, for the design and the deployment of reliable adaptive systems. SICODAF combines a behavioral reliability (absence of conflicting decisions) by means of transitions systems and an execution reliability (absence of inconsistencies) through a transactional middleware. SICODAF is based on autonomic computing. It allows to design and deploy an adaptive system in the form of an autonomic loop which consists of an abstraction layer, a transactional execution mechanism and a controller. SICODAF supports three types of controllers (based on rules, on continuous or discrete control theory). SICODAF also allows for loop reconfiguration, to deal with changing objectives in the considered system, and the integration of a hardware failure detection system. Finally, SICODAF allows for the design of multiple loops for systems that consist of a high number of entities or that require controllers of different types. These loops can be combined in parallel, coordinated or hierarchical.

SICODAF was implemented using the transactional middleware LINC, the abstraction environment PUTUTU and the language Heptagon/BZR that is based on transitions systems. SICODAF was also evaluated using two case studies.

Table des matières

1 Introduction **13**
- 1.1 Contexte . . . 13
- 1.2 Problématique . . . 14
- 1.3 Contribution . . . 15
- 1.4 Organisation du manuscrit . . . 16

2 État de l'art **19**
- 2.1 Systèmes considérés . . . 20
  - 2.1.1 Adaptation des systèmes . . . 20
  - 2.1.2 Fiabilité des systèmes . . . 21
    - 2.1.2.1 Fiabilité comportementale . . . 21
    - 2.1.2.2 Fiabilité d'exécution . . . 22
- 2.2 Conception et déploiement de systèmes adaptatifs fiables . . . 22
  - 2.2.1 Intergiciel . . . 22
  - 2.2.2 Calcul autonomique . . . 24
  - 2.2.3 Solutions proposées dans la littérature . . . 26
    - 2.2.3.1 Adaptation basée sur des règles . . . 26
    - 2.2.3.2 Adaptation basée sur une fonction d'utilité . . . 30
    - 2.2.3.3 Adaptation basée sur la théorie du contrôle . . . 31
  - 2.2.4 Solutions spécifiques au bâtiment intelligent . . . 34
    - 2.2.4.1 Interfaces graphiques de programmation . . . 35
    - 2.2.4.2 Effets des actionneurs sur l'environnement . . . 35
    - 2.2.4.3 Synthèse des solutions pour les bâtiments intelligents 35
  - 2.2.5 Conclusion . . . 36
- 2.3 Outils utilisés . . . 37
  - 2.3.1 LINC . . . 38
    - 2.3.1.1 Langage LINC . . . 38
    - 2.3.1.2 Déploiement des applications LINC . . . 40
  - 2.3.2 PUTUTU . . . 42
  - 2.3.3 Heptagon/BZR . . . 42
    - 2.3.3.1 Conception d'un programme H/BZR . . . 43
    - 2.3.3.2 Exécution d'un programme H/BZR . . . 47
  - 2.3.4 Conclusion . . . 47

3 Support Intergiciel et Conception d'une boucle autonomique fiable **49**
- 3.1 Présentation générale de SICODAF . . . 50
  - 3.1.1 Systèmes considérés . . . 50
    - 3.1.1.1 Applications considérées . . . 50
    - 3.1.1.2 Plateformes d'exécution considérées . . . 50
    - 3.1.1.3 Conception et déploiement des systèmes considérés . 51

3.1.2 Support intergiciel SICODAF 51
3.2 Conception d'une boucle générique 53
3.2.1 Définition de la classe de systèmes considérés 53
3.2.2 Flot de conception d'une boucle générique 54
3.2.2.1 Conception de la couche d'abstraction 55
3.2.2.2 Conception des règles d'observation et d'exécution . 55
3.2.2.3 Conception du contrôleur 57
3.2.3 Conception semi-automatique d'une boucle générique 62
3.2.3.1 Génération de la couche d'abstraction 62
3.2.3.2 Génération du contrôleur 62
3.3 Reconfiguration d'une boucle autonomique 63
3.3.1 Principe de la reconfiguration du contrôleur d'une boucle . . 63
3.3.2 Mise en œuvre de la reconfiguration d'une boucle 65
3.4 Intégration d'un système de détection de pannes 65
3.4.1 Exemple d'un système de détection de pannes 65
3.4.2 Mise en œuvre du système de détection de pannes 65
3.4.2.1 Ressources de calcul ou d'entrée/sortie avec une adresse IP 66
3.4.2.2 Ressources d'entrée/sortie sans adresse IP 66
3.4.3 Intégration du système de détection de pannes 67
3.5 Conclusion 67

4 Conception d'une boucle de déploiement et d'une boucle applicative 69
4.1 Conception d'une boucle de déploiement 69
4.1.1 Spécificité d'une boucle de déploiement 69
4.1.1.1 Données collectées par une boucle de déploiement . 69
4.1.1.2 Commandes d'une boucle de déploiement 69
4.1.1.3 Couche d'abstraction d'une boucle de déploiement . 71
4.1.2 Flot de conception d'une boucle de déploiement 72
4.1.2.1 Description du système considéré et de ses objectifs 72
4.1.2.2 Génération des entités de la couche d'abstraction . . 78
4.1.2.3 Génération de la fonction de transitions du contrôleur 78
4.1.2.4 Génération de la fonction de transitions 84
4.1.2.5 Génération de la règle qui exécute la fonction de transitions 85
4.1.3 Exemple de conception d'une boucle de déploiement 86
4.1.3.1 Exemple d'un système de traitement de données . . 86
4.1.3.2 Génération d'une boucle de déploiement 87
4.2 Conception d'une boucle applicative 89
4.2.1 Spécificité d'une boucle applicative de bâtiments intelligents . 89
4.2.1.1 Données collectées par une boucle applicative 89
4.2.1.2 Commandes d'une boucle applicative 90
4.2.1.3 Couche d'abstraction d'une boucle applicative . . . 90

4.2.2 Flot de conception d'une boucle applicative 91
4.2.2.1 Description du système et de ses objectifs 91
4.2.2.2 Génération des entités de la couche d'abstraction . . 92
4.2.2.3 Génération de la règle qui exécute la fonction de transitions 92
4.2.3 Exemple de conception d'une boucle applicative 93
4.2.3.1 Exemple de système d'éclairage 93
4.2.3.2 Description du système d'éclairage et son objectif . 93
4.2.3.3 Génération d'une boucle applicative 94
4.3 Conclusion 96

5 Support pour des boucles multiples 97
5.1 Motivations et avantages des boucles multiples 97
5.1.1 Motivations 98
5.1.2 Avantages 99
5.2 Modes de composition de boucles 100
5.3 Conception de boucles en parallèle 100
5.3.1 Avantages des boucles en parallèle 100
5.3.2 Limitations des boucles en parallèle 101
5.4 Conception de boucles coordonnées 102
5.4.1 Prérequis de la conception d'un coordinateur 102
5.4.2 Conception d'un coordinateur basé sur des règles 104
5.4.3 Conception d'un coordinateur basé sur le contrôle discret . . 105
5.4.3.1 Coordinateur validé par la vérification formelle . . . 105
5.4.3.2 Coordinateur basé sur la synthèse de contrôleurs discrets 106
5.4.4 Limitations des boucles coordonnées 107
5.5 Conception de boucles hiérarchiques 107
5.5.1 Conception à l'aide du langage de règles 108
5.5.2 Conception à l'aide de la théorie du contrôle discret 108
5.5.2.1 Validation à l'aide de la vérification formelle 109
5.5.2.2 Conception à l'aide de la synthèse de contrôleurs discrets 109
5.6 Conclusion 110

6 Mise en œuvre et Études de cas 111
6.1 Implémentation 111
6.1.1 Combinaison de LINC et Heptagon/BZR 112
6.1.1.1 Invocation de la fonction *step* 112
6.1.1.2 Encapsulation de la fonction *Step* 115
6.1.2 Implémentation de boucles de déploiement 116
6.1.2.1 Implémentation de la couche d'abstraction 116
6.1.2.2 Langage de description textuel 118

6.1.2.3 Générateur de règles exécutant une fonction de transitions . . . 118
6.1.3 Implémentation de boucles applicatives . . . 119
6.1.3.1 Générateur de couches d'abstraction . . . 119
6.1.3.2 Générateur de règles *templates* . . . 119
6.1.3.3 Générateur de règles instances . . . 120
6.2 Études de cas . . . 120
6.2.1 Déploiement d'un système de traitement de données . . . 120
6.2.1.1 Description des entités du système . . . 120
6.2.1.2 Description des objectifs du système . . . 121
6.2.1.3 Mise en œuvre d'une boucle de déploiement . . . 121
6.2.1.4 Comportement de la boucle . . . 125
6.2.2 Conception d'une pièce de bureau intelligente . . . 127
6.2.2.1 Description des capteurs et des actionneurs . . . 127
6.2.2.2 Description des objectifs . . . 128
6.2.2.3 Mise en œuvre d'une boucle applicative . . . 129
6.3 Conclusion . . . 138

7 Conclusion et Perspectives **141**
7.1 Conclusion . . . 141
7.1.1 Rappel du contexte et de la problématique . . . 141
7.1.2 Contribution de la thèse . . . 141
7.2 Perspectives . . . 142
7.2.1 SICODAF pour d'autres domaines ou problématiques . . . 142
7.2.2 Extension de SICODAF . . . 143
7.2.2.1 Améliorations de la boucle générique . . . 143
7.2.2.2 Conception automatique de boucles . . . 143

Bibliographie **145**

Table des figures

1.1 Architecture d'un système autonomique 14
1.2 Architecture d'une boucle générique 16

2.1 Architecture d'un système autonomique 26
2.2 Boucle de contrôle continu 32
2.3 Système de transitions 34
2.4 Système de transitions contrôlé 34
2.5 Environnement d'abstraction PUTUTU 43
2.6 Automate modélisant le contrôle d'une ressource de calcul 44
2.7 Exemple de nœud avec *contrat* 46
2.8 Exemple de nœud modulaire 46

3.1 Architecture d'un système autonomique 52
3.2 Architecture d'une boucle autonomique fiable 52
3.3 Méta-modèle des applications considérées 54
3.4 Méta-modèle des plateformes d'exécution considérées 54
3.5 Structure d'un contrôleur 57
3.6 Principe de reconfiguration d'une boucle autonomique 64

4.1 Génération d'une boucle autonomique 73
4.2 Mémoires associatives de la couche d'abstraction 78
4.3 Automate de la tache T1 79
4.4 Automate de l'objet O1 80
4.5 Automate de la règle R1 81
4.6 Automate de la ressource Octave 82
4.7 Automate de l'hôte de la ressource de calcul Rsc1 83
4.8 Automate de la ressource de calcul D1 83
4.9 *Contrat* du système de traitement de données 88
4.10 *Contrat* du système de traitement de données 88
4.11 Description d'un actionneur de lampe 94
4.12 Description d'un actionneur de volet 94
4.13 Description d'un actionneur de volet 95

5.1 Boucles en parallèle 101
5.2 Structure d'une étage de l'exemple du bâtiment B_1 102
5.3 Boucles coordonnées 103
5.4 Contrôle du contrôleur de la boucle de `Piece1` 105
5.5 Contrôle du contrôleur de la boucle de `Piece2` 106
5.6 Coordinateur du couloir `C1` par vérification formelle 106
5.7 Coordinateur du couloir `C1` par synthèse de contrôleurs discrets . . . 107

5.8 Boucles hiérarchiques 108
5.9 Spécification du contrôleur `ctrl1` 109
5.10 Spécification du contrôleur `ctrl4` 109

6.1 Automate de l'imprimante $P1$ 122
6.2 *Contrat* du système de traitements de données 124
6.3 Chronogramme du système de traitements de données 126
6.4 Automate d'une porte 133
6.5 Automate d'un climatiseur réversible 133
6.6 Automate d'une ventilation mécanique 134
6.7 Automate d'une fenêtre 134
6.8 Nœud H/BZR avec le *contrat* de la pièce $Piece1$ 135
6.9 Démonstrateur 137

Liste des tableaux

2.1 Solutions de détection d'erreurs entre des règles 29
2.2 Solutions de conception et de déploiement de systèmes adaptatifs . . 37

6.1 Application de traitement de données 121
6.2 Plateforme d'exécution de l'application de traitement de données . . 121
6.3 Comparaison des temps de synthèse de contrôleurs 138

Chapitre 1

Introduction

1.1 Contexte

Aujourd'hui, avec l'informatique pervasive et l'internet des objets, les systèmes (p. ex., bâtiment intelligent, contrôle de procédés industriels) sont hétérogènes et distribués. Ils sont constitués de nombreuses entités matérielles (p. ex., capteurs, actionneurs, ressources de calcul) et logicielles (p. ex., bases de données) qui sont réparties dans différents endroits et qui interagissent pour réaliser un certain nombre d'objectifs (p. ex., économie d'énergie, fonctionnement continu). De tels systèmes opèrent dans des environnements dynamiques qui peuvent faire l'objet de plusieurs changements (p. ex., variations de charges, panne d'une ressource de calcul). Pour réaliser leurs objectifs et rester opérationnels, ces systèmes s'adaptent aux changements qui surviennent dans leur environnement et sont appelés systèmes adaptatifs.

Un système adaptatif est un système qui a la capacité de modifier son comportement en réponse aux changements de son environnement [99]. Pour ce faire, le système collecte des données à l'aide de capteurs, analyse les données collectées, prend des décisions et exécute les actions correspondantes à l'aide d'actionneurs.

Les systèmes distribués adaptatifs peuvent être conçus à l'aide d'intergiciels adaptatifs [111, 31]. Ces intergiciels permettent de gérer les aspects liés à la distribution (communications entre entités distantes) et fournissent un support pour l'adaptation des systèmes. Certains de ces intergiciels (p. ex., SOCAM [54], Facts [128], LINC [79]) utilisent un langage à base de règles et permettent de concevoir un système en spécifiant les actions à effectuer lorsque des événements spécifiques surviennent.

Dans un système adaptatif, la logique d'adaptation doit être séparée de la logique du système à adapter. Cela permet la réutilisation de la logique d'adaptation, pour un autre système, et son évolution. Une approche qui permet une séparation nette entre la logique d'adaptation et le système à adapter est le calcul autonomique [64]. Comme illustré à la Figure 2.1, le système est constitué d'un gestionnaire autonomique qui gère un ensemble d'entités pouvant être logicielles et/ou matérielles, sous la forme d'une boucle autonomique. Le gestionnaire autonomique observe les entités gérées en collectant des données. Ensuite il analyse les données collectées pour planifier et exécuter des actions, sur la base d'une connaissance sur les entités.

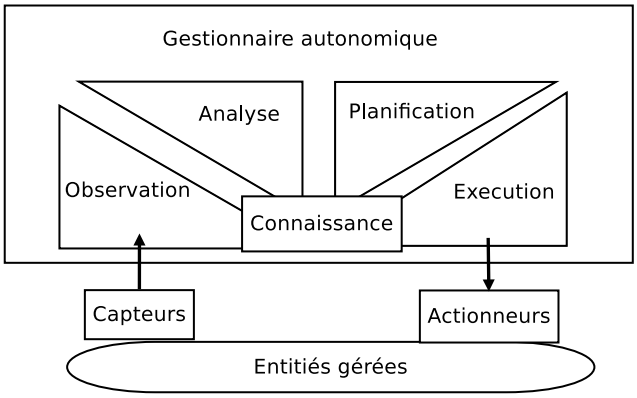

FIGURE 1.1 – Architecture d'un système autonomique

1.2 Problématique

Concevoir et déployer un système adaptatif consistent à mettre en place une boucle autonomique qui gère les entités du système pour réaliser les objectifs fixés. Par exemple, considérons une plateforme d'exécution composée de trois ressources de calcul sur lesquelles sont exécutées des entités logicielles. Lorsque la charge de travail est faible, la boucle autonomique peut décider de regrouper les entités logicielles sur une seule ressource de calcul et éteindre les deux autres, dans le but de réduire la quantité d'énergie consommée par la plate forme d'exécution. Par exemple dans un bâtiment intelligent, lorsqu'une présence est détectée dans une pièce, la boucle autonomique peut allumer la lampe ou ouvrir le volet pour éclairer la pièce.

La conception d'un système adaptatif pose plusieurs problèmes. D'abord, les capteurs et les actionneurs sont produits par différents fabricants et utilisent des technologies de communication différentes. De même, les ressources de calcul sont hétérogènes. Elles sont de différents types (p. ex., PC, Raspberry Pi), ont des capacités, en termes de CPU et de RAM, différentes, des systèmes d'exploitation différents et des modes de démarrage différents. Certaines ressources de calcul (p. ex., PC) peuvent être démarrées, à travers le réseau, en utilisant la technique du Wake on LAN (WoL) [1]. D'autres ressources de calcul (p. ex., Raspberry Pi) ne peuvent pas être démarrées avec le WoL parce qu'elles doivent être débranchées puis rebranchées.

Ensuite, l'exécution des actions qui correspondent aux décisions prises peut créer des inconsistances. En effet, une ressource de calcul peut tomber en panne ou devenir inaccessible à cause d'une erreur de communication. Dans les deux cas, la ressource de calcul ne reçoit pas la commande qui lui a été envoyée et l'action correspondante n'est pas effectuée. Le fait de supposer que l'action a été effectuée crée une inconsistance. Par exemple, démarrer une ressource de calcul, pour y déployer une entité logicielle, et supposer qu'elle est démarrée devient une inconsistance si la ressource de calcul reste éteinte du fait d'une panne. Dans ce cas, le déploiement est supposé être effectué mais ne l'est pas. Un exemple, d'inconsistance dans le domaine du bâ-

1. http ://openclassrooms.com/courses/wake-on-lan

timent intelligent est d'allumer une lampe et de supposer qu'elle est allumée alors qu'elle est restée éteinte du fait d'une panne ou d'une erreur de communication.

Ensuite, les décisions prises pour l'adaptation du système peuvent être conflictuelles. Elles peuvent aussi violer un ou plusieurs objectifs. Par exemple, la boucle autonomique peut décider de migrer plusieurs entités logicielles sur une ressource de calcul qui est éteinte ou qui n'est pas disponible pour des raisons de maintenance.

De plus, la conception d'un système adaptatif requiert une séparation des aspects déploiement et applicatif, par la mise en place de deux boucles autonomiques : une boucle de déploiement et une boucle applicative. La mise en œuvre de ces boucles nécessite une aide à la conception par un langage de spécification de haut niveau.

Enfin, selon la nature du système considéré, il peut être nécessaire de mettre en œuvre plusieurs boucles applicatives et/ou de déploiement. Par exemple, pour un système constitué d'un grand nombre d'entités, utiliser une seule boucle pour gérer toutes les entités peut causer des dégradations de performance. Ces boucles doivent être coordonnées pour éviter les décisions conflictuelles et les violations d'objectifs.

1.3 Contribution

La contribution de cette thèse est un support intergiciel, appelé SICODAF (Support Intergiciel pour la COnception et le Déploiement Adaptatifs Fiables), pour la conception et le déploiement de systèmes adaptatifs fiables. Ce support intergiciel permet de concevoir et de déployer un système adaptatif sous la forme d'une boucle qui combine deux formes de fiabilité : une fiabilité comportementale et une fiabilité d'exécution. La fiabilité comportementale consiste à prendre des décisions d'adaptation correctes et cohérentes. Elle garantit l'absence de conflits et de violations d'objectifs et est obtenue à l'aide de systèmes de transitions. La fiabilité d'exécution consiste à détecter le fait qu'une action ne peut pas être effectuée du fait d'une erreur de communication ou d'une panne matérielle. Elle garantit l'absence d'inconsistances à l'exécution et est obtenue au moyen d'un intergiciel transactionnel.

La Figure 3.2 présente l'architecture d'une boucle autonomique mise en œuvre à l'aide du support intergiciel SICODAF. Cette boucle est constituée d'une couche d'abstraction, d'un mécanisme d'exécution transactionnelle et d'un contrôleur. La couche d'abstraction communique avec les entités du système et masque leur hétérogénéité. Le mécanisme d'exécution transactionnelle exécute des règles d'observation et des règles d'exécution. Ces règles collectent des données sur le système et exécutent des commandes. Le contrôleur analyse les données collectées et calcule les commandes à exécuter, sur la base d'une connaissance sur le système. Le contrôleur peut être basé sur des règles, sur la théorie du contrôle continu [52] ou du contrôle discret [25]. Cette boucle peut être, selon la nature des entités du système à adapter, une boucle de déploiement ou une boucle applicative. Elle est une boucle de déploiement lorsqu'elle gère des entités logicielles et des ressources de calcul pour réaliser des objectifs tels que le fonctionnement continu et l'économie d'énergie de la plateforme d'exécution. Une telle boucle est une boucle applicative, par exemple dans

le contexte du bâtiment intelligent, lorsqu'elle contrôle les actionneurs du bâtiment pour réaliser des objectifs tels que la régulation de la température et l'éclairage.

SICODAF permet la reconfiguration d'une boucle, pour gérer les changements d'objectifs qui surviennent dans système à adapter, et l'intégration d'un système de détection de pannes. SICODAF permet également, pour la conception et le déploiement d'un système composé de nombreuses entités, la mise en œuvre de boucles multiples. Ces boucles peuvent être composées en utilisant le mode de composition approprié pour minimiser les coûts de conception et d'exécution. Les modes de composition supportés par SICODAF sont : parallèle, coordonné et hiérarchique.

Le support intergiciel SICODAF a été mis en œuvre en utilisant l'intergiciel à base de règles transactionnelles LINC [79], l'environnement d'abstraction PUTUTU [100] et le langage Heptagon/BZR [36] basé sur des systèmes de transitions.

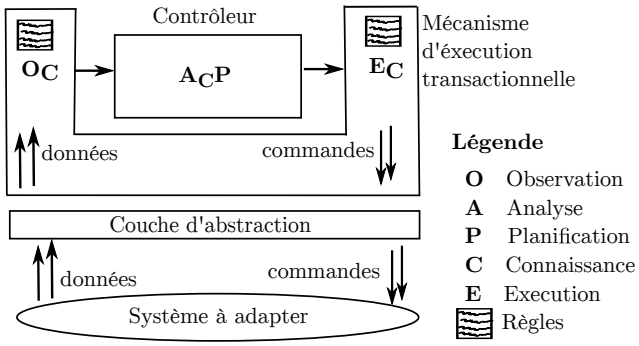

FIGURE 1.2 – Architecture d'une boucle générique

1.4 Organisation du manuscrit

Le manuscrit est divisé en sept chapitres. Le **chapitre 2** présente l'état de l'art sur la conception et le déploiement des systèmes adaptatifs. Il rappelle d'abord le contexte et la problématique de la thèse. Ensuite, il présente les solutions proposées dans la littérature en se focalisant sur celles basées sur des intergiciels et/ou sur le calcul autonomique. Enfin, ce chapitre présente dans le domaine applicatif du bâtiment intelligent, les problèmes rencontrés et les solutions qui ont été proposées.

Le **chapitre 3** présente une vue d'ensemble du support intergiciel proposé, SICODAF, et la conception d'une boucle générique. Il décrit d'abord SICODAF et présente le flot de conception d'une boucle générique. Ensuite, il présente la reconfiguration d'une boucle et l'intégration d'un système de détection de pannes.

Le **chapitre 4** présente d'abord la conception d'une boucle de déploiement. Ensuite, il décrit la conception d'une boucle applicative de bâtiment intelligent.

Le **chapitre 5** décrit le support pour des boucles multiples. Il présente d'abord les motivations et avantages de la conception de boucles multiples. Ensuite, il présente trois modes de composition de boucles (boucles en parallèle, boucles coordonnées et boucles hiérarchiques) et leur conception à l'aide du support SICODAF.

Le **chapitre 6** présente l'implémentation puis la validation du support intergiciel SICODAF. Pour la validation, deux études de cas sont d'abord présentées puis des boucles sont conçues pour permettre leur conception et leur déploiement.

Le **chapitre 7** conclut le manuscrit et présente les perspectives de la thèse.

Chapitre 2

État de l'art

Sommaire

2.1 Systèmes considérés 20
2.1.1 Adaptation des systèmes 20
2.1.2 Fiabilité des systèmes 21
2.1.2.1 Fiabilité comportementale 21
2.1.2.2 Fiabilité d'exécution 22
2.2 Conception et déploiement de systèmes adaptatifs fiables . 22
2.2.1 Intergiciel 22
2.2.2 Calcul autonomique 24
2.2.3 Solutions proposées dans la littérature 26
2.2.3.1 Adaptation basée sur des règles 26
2.2.3.2 Adaptation basée sur une fonction d'utilité 30
2.2.3.3 Adaptation basée sur la théorie du contrôle 31
2.2.4 Solutions spécifiques au bâtiment intelligent 34
2.2.4.1 Interfaces graphiques de programmation 35
2.2.4.2 Effets des actionneurs sur l'environnement 35
2.2.4.3 Synthèse des solutions pour les bâtiments intelligents 35
2.2.5 Conclusion 36
2.3 Outils utilisés 37
2.3.1 LINC 38
2.3.1.1 Langage LINC 38
2.3.1.2 Déploiement des applications LINC 40
2.3.2 PUTUTU 42
2.3.3 Heptagon/BZR 42
2.3.3.1 Conception d'un programme H/BZR 43
2.3.3.2 Exécution d'un programme H/BZR 47
2.3.4 Conclusion 47

Ce chapitre présente l'état de l'art sur la conception et le déploiement de systèmes adaptatifs fiables. Il introduit d'abord les systèmes adaptatifs et leur fiabilité. Ensuite, il présente les solutions proposées, dans la littérature, pour la conception et le déploiement de systèmes adaptatifs fiables. Enfin, ce chapitre présente les outils qui seront utilisés, par la suite, pour la mise en œuvre du support intergiciel proposé.

2.1 Systèmes considérés

Dans le contexte de l'informatique pervasive et de l'internet des objets [7], les systèmes sont constitués de nombreuses entités matérielles (p. ex., ressources de calcul, capteurs, actionneurs) et logicielles (p. ex., bases de données). Des exemples de tels systèmes sont les bâtiments intelligents, les systèmes de gestion des transports, les systèmes de soins médicaux et les systèmes de contrôle de procédés industriels.

Dans ces systèmes, différentes entités coopèrent dans le but de réaliser un certain nombre d'objectifs. Par exemple, un bâtiment intelligent est équipé de nombreux capteurs, d'actionneurs et de ressources de calcul qui sont contrôlés, de façon automatique, par un ensemble d'entités logicielles. L'objectif est, par exemple, d'assurer le confort des occupants tout en minimisant la consommation d'énergie du bâtiment.

Ces systèmes opèrent dans des environnements dynamiques et sont à la fois

— **hétérogènes** : les ressources de calcul qui constituent ces systèmes sont de différents types et ont des capacités de calcul, de stockage et de communication différentes. De plus, les ressources de calcul ont des systèmes d'exploitation différents et des modes de démarrage différents. Certaines ressources de calcul (p. ex., PC) peuvent être démarrées à travers le réseau en utilisant la technique du Wake on LAN (WoL). D'autres ressources de calcul (p. ex., Raspberry Pi) ne peuvent pas être démarrées en utilisant le WoL parce qu'elles doivent être débranchées et rebranchées pour démarrer. De même, les capteurs et les actionneurs sont hétérogènes. Ils sont produits par différents fabricants et utilisent des technologies de communication qui sont différentes (p. ex., ZigBee [139], EnOcean [42], Plugwise [105], MODBUS [90]) ;

— **distribués** : les entités qui constituent ces systèmes sont physiquement distribuées et interagissent à travers le réseau pour réaliser les objectifs fixés. Les motivations de la distribution de ces systèmes sont, par exemple, des raisons de performance (p. ex., plus de capacité de calcul), des raisons de tolérance aux pannes (p. ex., redondance d' entités logicielles) ou des contraintes géographiques. Les interactions entre les différentes entités requièrent la résolution de problèmes tels que le partage de données, les accès concurrents et la synchronisation. De plus, les entités qui constituent ces systèmes peuvent devenir inaccessibles à cause d'une erreur de communication ou d'une panne.

Pour rester opérationnels et réaliser leurs objectifs, ces systèmes doivent s'adapter aux changements de leur environnement (p. ex., panne d'une entité matérielle).

2.1.1 Adaptation des systèmes

L'adaptation, pour des systèmes, consiste à modifier leur comportement en réponse aux changements qui surviennent dans leur environnement [27, 99]. L'environnement étant tout ce qui peut être perçu par les systèmes. Du fait du nombre élevé d'entités qui constituent ces systèmes et de la nature dynamique de leur environnement, l'adaptation manuelle n'est pas réalisable. Les systèmes doivent s'adapter de façon autonome et sont appelés systèmes auto-adaptatifs ou, seulement, adaptatifs.

Pour s'adapter, un système collecte de façon continue des données de son environnement. Le but est de détecter les changements qui surviennent. Ensuite, le système analyse les données collectées pour prendre des décisions et exécuter les actions correspondantes. Par exemple, dans le contexte du bâtiment intelligent, le système peut allumer le chauffage d'une pièce lorsqu'une présence y est détectée et que la température est inférieure à 17 ° C. Lorsque la charge de travail est faible, le système peut regrouper toutes les entités logicielles sur une ressource de calcul et éteindre les autres ressources de calcul pour réduire la consommation d'énergie.

Pour éviter que les systèmes se retrouvent dans des états qui sont indésirables, ne pouvant plus réaliser leurs objectifs, l'adaptation doit être effectuée de façon fiable.

2.1.2 Fiabilité des systèmes

La fiabilité d'un système est définie dans [11] comme étant la continuité du fonctionnement correct. Elle consiste à éviter les états indésirables qui compromettent le bon fonctionnement du système. Par exemple, dans le contexte du bâtiment intelligent, la non fiabilité d'un système peut causer des comportements indésirables ou anormaux pouvant frustrer les occupants du bâtiment. Des exemples de tels comportements sont : des volets qui s'ouvrent et se ferment en boucle, une pièce occupée qui n'est pas éclairée ou la température qui est supérieure à 30 ° C dans un bureau occupé. Ces comportements doivent être évités pour garantir la fiabilité du système.

La fiabilité d'un système adaptatif dépend des décisions prises, pour réagir aux changements survenus dans l'environnement, et de l'exécution des actions correspondantes. Elle consiste en une fiabilité comportementale et une fiabilité d'exécution.

2.1.2.1 Fiabilité comportementale

La fiabilité comportementale consiste à prendre des décisions d'adaptation qui sont à la fois correctes et cohérentes. En effet, les décisions prises par un système pour s'adapter aux changements de son environnement peuvent être conflictuelles. Elles peuvent aussi violer un ou plusieurs objectifs du système. Les conflits et les violations d'objectifs sont explicites ou implicites. Ils sont implicites lorsqu'ils sont dus aux effets de l'exécution des actions qui correspondent aux décisions d'adaptation prises.

Par exemple, dans le contexte du bâtiment intelligent, le fait d'ouvrir une fenêtre pour ventiler une pièce peut violer un objectif qui limite le niveau de bruit de la pièce à un certain seuil. De même, migrer une entité logicielle d'une ressource de calcul à une autre peut violer un objectif relatif à la performance qui consiste à ne pas surcharger une ressource de calcul. Enfin, migrer une ou plusieurs entités logicielles vers une ressource de calcul qui n'est pas allumée est une décision conflictuelle.

Pour la fiabilité des systèmes, les conflits et les violations d'objectifs doivent être évités lors de la prise de décisions. De plus, les actions qui correspondent aux décisions prises doivent être correctement exécutées, en prenant en compte le fait que des erreurs de communication et des pannes matérielles peuvent survenir.

2.1.2.2 Fiabilité d'exécution

La fiabilité d'exécution consiste à éviter les inconsistances qui peuvent être causées par des erreurs de communication et des pannes matérielles. Une inconstance survient lorsque le système suppose qu'une action est effectuée alors qu'elle ne l'est pas en réalité du fait de l'occurrence d'une erreur de communication ou d'une panne.

Par exemple, le fait d'ouvrir une fenêtre pour ventiler une pièce crée une inconsistance si le système suppose que la fenêtre a été ouverte alors qu'elle ne l'est pas du fait d'une erreur de communication. De même, démarrer une ressource de calcul et y déployer des entités logicielles crée une inconsistance si la ressource de calcul reste éteinte du fait d'une panne. Dans ce cas, les entités logicielles sont supposées être déployées alors qu'elles ne le sont pas, ce qui peut causer l'arrêt du système.

2.2 Conception et déploiement de systèmes adaptatifs fiables

Du fait de la nature distribuée des systèmes adaptatifs, plusieurs solutions proposées, dans la littérature, pour leur conception et leur déploiement utilisent un intergiciel. Cela permet de gérer la communication entre les entités distribuées et leurs interactions. Pour permettre l'adaptation autonome des systèmes, plusieurs solutions, de conception et de déploiement, sont basées sur le calcul autonomique. Les intergiciels, le calcul autonomique et les solutions de conception et de déploiement de systèmes adaptatifs proposées dans la littérature sont présentés ci-après.

2.2.1 Intergiciel

Un intergiciel est un outil logiciel qui simplifie la conception de systèmes distribués. Il fournit des mécanismes d'abstraction qui masquent l'hétérogénéité des systèmes et permettent la communication ainsi que la coordination des entités distribuées [41]. Ces mécanismes permettent aux développeurs de se concentrer sur la logique du système (les fonctionnalités à fournir et les objectifs à atteindre) et non sur les aspects liés à sa distribution (p. ex., les interactions entre entités distantes).

De plus, certains intergiciels fournissent des mécanismes qui permettent aux systèmes d'avoir une connaissance de leur contexte d'exécution, en collectant et en interprétant des données dans le but de détecter des changements. Plusieurs intergiciels ont été proposés dans le contexte de l'informatique pervasive et de l'internet des objets. Ces intergiciels utilisent différentes approches pour permettre la conception de systèmes distribués et peuvent être classés, d'après [109], en sept catégories :

1. **Intergiciels orientés événements** : dans un intergiciel orienté événements, les entités du système considéré interagissent à travers des événements. Ces événements sont produits par des entités appelées producteurs et sont consommés par d'autres entités appelées consommateurs. Ces intergiciels sont généralement basés sur le paradigme publier/souscrire (« *publish/suscribe* ») [44]. Les consommateurs souscrivent aux événements qui les intéressent et n'ont

pas besoin de connaître leurs producteurs ou d'être exécutés au même instant qu'eux. De même, les producteurs n'ont pas besoin de connaître les consommateurs. Ces intergiciels permettent le découplage spatial et temporel des entités du système considéré. Ce découplage permet de gérer la nature dynamique des systèmes (une entité matérielle peut tomber en panne ou devenir disponible). Des exemples de tels intergiciels sont Hermes [104] et Green [120].

2. **Intergiciels orientés services** : dans un intergiciel orienté services, des entités, appelées fournisseurs de services, fournissent des services qui sont découverts et utilisés par d'autres entités appelées consommateurs de services. Ces intergiciels sont basés sur l'architecture orientée services et bénéficient des avantages tels que l'interopérabilité, le couplage faible entre les différents services et la possibilité de découvrir et de composer des services. Des exemples d'intergiciels orientés services sont MUSIC [110] et SOCRADES [57].
3. **Intergiciels orientés agents** : dans un intergiciel orienté agents, l'application considérée est conçue sous la forme de programmes modulaires qui sont distribués entre les entités matérielles du système (p. ex., ressources de calcul) à l'aide d'agents mobiles. Un agent mobile peut migrer d'une entité à une autre et maintient son état d'exécution lors de la migration. Les intergiciels orientés agents facilitent la conception de systèmes distribués capables de tolérer des pannes partielles, du fait de la mobilité des agents. Cependant, la nature autonome des agents peut causer des comportements imprévisibles à l'exécution. Des exemples de tels intergiciels sont Impala [77] et Agilla [47].
4. **Intergiciels à base de machines virtuelles** : dans un intergiciel à base de machines virtuelles, l'application considérée est conçue en modules séparés qui sont distribués entre les entités matérielles du système. Chaque entité matérielle possède une machine virtuelle pour l'interprétation des différents modules qui lui sont alloués. Ces intergiciels supportent l'hétérogénéité des entités matérielles qui constituent le système, grâce à la virtualisation. Cependant, la virtualisation requiert d'importantes ressources en termes de CPU et de RAM et peut ne pas être faisable sur toutes les entités matérielles du système. Des exemples de tels intergiciels sont Maté [75] et Melete [136].
5. **Intergiciels orientés bases de données** : dans un intergiciel orienté bases de données, un réseau de capteurs est considéré comme un système de base de données relationnelle virtuelle. Une application peut interroger la base de données à l'aide d'un langage de requêtes (p. ex., SQL) pour obtenir des données des capteurs. Ces intergiciels simplifient l'accès aux données mais utilisent une approche centralisée qui limite le passage à l'échelle. Des exemples d'intergiciels orientés bases de données sont TinyDB [81] et Sensation [58].
6. **Intergiciels à application spécifique** : ces intergiciels sont conçus pour répondre aux besoins spécifiques d'une application ou d'un domaine d'application (p. ex., bâtiments intelligents, transports). Ces intergiciels sont limités par le fait qu'ils n'offrent pas de support pour plusieurs domaines d'application. Des exemples de tels intergiciels sont MidFusion [1] et TinyCubus [85].

7. **Intergiciels à base de tuples** : dans un intergiciel à base de tuples, les entités du système considéré interagissent à travers des mémoires associatives (« *associative memories* ») qui contiennent des données sous forme de tuples. Ces intergiciels sont dérivés du langage de coordination Linda [24]. Ils fournissent une interface de programmation applicative simple. Cette interface permet d'accéder aux mémoires associatives pour lire, consommer ou insérer des tuples. De plus, ces intergiciels permettent le découplage spatial et temporel entre les différentes entités du système considéré (les entités n'ont pas besoin de se connaître ni d'exister au même instant). Des exemples de tels intergiciels sont TOTA [84], EgoSpaces [62], TuCSoN [98], LIME [95], MARS [20], MobiGATE [138], Holoparadigm [13], SAPERE [26] et LINC [79].

Un intergiciel peut appartenir à plusieurs catégories et bénéficier de leurs avantages. Par exemple, un intergiciel orienté agents peut utiliser des mémoires associatives et permettre le découplage spatial et temporel des différents agents. De même, un intergiciel à base de tuples peut utiliser une approche orientée agents. Cela permet, grâce à la mobilité des agents, l'adaptation des systèmes distribués. Une approche pour rendre l'adaptation autonome est le calcul autonomique [64].

2.2.2 Calcul autonomique

Le calcul autonomique [64] a été introduit en 2001 par IBM pour rendre autonome l'administration des systèmes informatiques. Il permet la conception de systèmes capables de s'adapter à leur environnement de façon autonome en effectuant

- **l'auto-réparation** : consiste à détecter, diagnostiquer et réparer les dysfonctionnements causés par des pannes matérielles et des erreurs logicielles ;
- **l'auto-optimisation** : consiste à, continuellement, améliorer son fonctionnement (p. ex., en termes de performance et de consommation de ressources) ;
- **l'auto-protection** : consiste à se défendre contre les attaques malicieuses ou les dysfonctionnements qui n'ont pas pu être résolus par l'auto-réparation ;
- **l'auto-configuration** : consiste à se reconfigurer, de façon automatique, en fonction des changements qui surviennent dans le but de rester opérationnel.

Un système capable de s'adapter, aux changements qui surviennent dans son environnement de façon autonome, en effectuant l'auto-réparation, l'auto-optimisation, l'auto-configuration ou l'auto-protection est appelé système autonomique. Un tel système est caractérisé par une séparation nette entre la logique d'adaptation et le système à adapter. Un tel système est, comme illustré à la Figure 2.1, constitué

- **d'entités gérées** : elles peuvent être matérielles (p. ex., ressources de calcul) et/ou logicielles (p. ex., logiciels de traitement d'images) et constituent le système à adapter pour réaliser un ensemble d'objectifs (p. ex., performance) ;
- **de capteurs** : ils permettent d'observer le système à adapter et son environnement. Un capteur peut être matériel (p. ex., capteur de température) ou logiciel (p. ex., entité logicielle conçue pour détecter d'une panne matérielle) ;
- **d'actionneurs** : ils permettent d'effectuer des actions sur le système (p. ex., démarrer une ressource de calcul) et peuvent être matériels ou logiciels ;

— **d'un gestionnaire autonomique** : il communique avec les capteurs et les actionneurs et est chargé de contrôler le comportement du système à adapter.

Le gestionnaire autonomique fonctionne sous la forme d'une boucle, appelée MAPE-K [64], en effectuant quatre opérations : *Observation* (« *Monitoring* »), *Analyse* (« *Analysis* »), *Planification* (« *Planning* ») et *Exécution* (« *Execution* »), sur la base d'une connaissance sur le système (« *Knowledge* »). Le gestionnaire autonomique observe, de façon continue, le système à adapter en collectant des données des capteurs (*Observation*). Ensuite, il analyse les données collectées pour détecter les changements qui sont survenus (*Analyse*) et prendre des décisions d'adaptation (*Planification*). Enfin, il effectue les actions correspondantes à l'aide des actionneurs (*Exécution*) et met à jour la connaissance qu'il a sur le système. L'analyse et la planification sont essentielles. Elles permettent de détecter le fait que le système doit s'adapter et de décider des actions à effectuer pour mettre en œuvre l'adaptation.

L'adaptation d'un système peut être effectuée par un ou plusieurs gestionnaires autonomiques. Lorsque l'adaptation est effectuée par un seul gestionnaire autonomique, l'architecture du système est dite centralisée. Cette architecture est adoptée par plusieurs solutions de conception et de déploiement de systèmes adaptatifs. Des exemples de telles solutions sont RAINBOW [49], QoSMOS [21], DACAR [38] et MADME [33]. Dans ces solutions, un gestionnaire autonomique effectue les opérations d'observation, d'analyse, de planification et d'exécution sur le système à adapter. L'avantage est que le gestionnaire autonomique a une vision globale du système et peut prendre des décisions d'adaptation appropriées. Cependant, lorsque le système considéré est constitué de nombreuses entités, l'utilisation d'un seul gestionnaire autonomique peut causer, à l'exécution, des dégradations de performance.

Pour éviter les dégradations de performance à l'exécution, certaines solutions de conception et de déploiement de systèmes adaptatifs proposées dans la littérature adoptent une architecture décentralisée. Des exemples de telles solutions sont celles proposées dans [133, 135]. Dans [133], plusieurs gestionnaires autonomiques sont utilisés pour permettre l'adaptation d'un système. Chaque gestionnaire autonomique effectue les opérations d'*Observation*, d'*Analyse*, de *Planification* et d'*Execution* (*OAPE)* sur un sous-système. Dans certains cas, pour prendre une décision, la planification est effectuée par plusieurs gestionnaires autonomiques qui interagissent par échange de messages. Dans [133], les auteurs proposent des modes d'interactions de gestionnaires autonomiques (p. ex., le mode hiérarchique). Dans le mode hiérarchique, les gestionnaires autonomiques sont structurés sous la forme d'une hiérarchie. Les gestionnaires autonomiques qui possèdent le niveau hiérarchique le plus faible effectuent les opérations *OAPE* sur le système à adapter et peuvent recevoir, en entrée, des données de la part des gestionnaires du niveau hiérarchique supérieur.

Le calcul autonomique est utilisé par plusieurs solutions de conception et de déploiement de systèmes adaptatifs pour bénéficier de la nature autonome de l'adaptation. Plusieurs autres solutions permettant l'adaptation autonome des systèmes, sans se référer au calcul autonomique ou le mentionner de façon explicite, ont été également proposées. Ces différentes solutions, proposées dans la littérature, pour la conception et le déploiement de systèmes adaptatifs sont présentées par la suite.

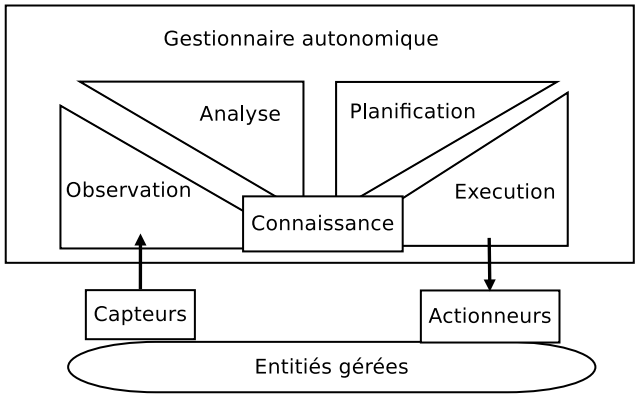

FIGURE 2.1 – Architecture d'un système autonomique

2.2.3 Solutions proposées dans la littérature

De nombreuses solutions ont été proposées, dans la littérature, pour permettre la conception et le déploiement de systèmes adaptatifs tout en garantissant leur fiabilité. Ces solutions sont basées sur des règles, des fonctions d'utilité, des objectifs ou des modèles [69]. Les règles, les fonctions d'utilité et les objectifs sont utilisés, par les solutions, pour prendre des décisions d'adaptation et peuvent être combinés avec des modèles qui représentent les comportements ou les structures des systèmes.

Dans les solutions basées sur des règles (p. ex., [96, 83, 71, 66, 119, 14, 32, 63, 38]), l'adaptation consiste à déclencher une ou plusieurs règles qui effectuent des actions lorsqu'un événement spécifique survient. Dans les solutions basées sur une fonction d'utilité (p. ex., [65, 89, 110, 50, 23]), l'adaptation consiste à choisir la configuration du système qui offre la plus grande utilité. Enfin, dans les solutions basées sur des objectifs (p. ex., [112, 118, 33, 88, 68]), l'adaptation consiste à réaliser les objectifs du système. Dans ces solutions, les objectifs sont réalisés à l'aide d'algorithmes spécifiques, à l'aide de la résolution de contraintes ou grâce à la théorie du contrôle.

Par la suite, sont présentées dans les détails l'adaptation basée sur des règles, l'adaptation basée sur des fonctions d'utilité et l'adaptation basée sur la théorie du contrôle, qui sont utilisées par de nombreuses solutions proposées dans la littérature.

2.2.3.1 Adaptation basée sur des règles

Elle consiste à utiliser un ensemble de règles qui spécifient les actions que le système considéré doit effectuer lorsque des événements spécifiques surviennent. Les règles sont généralement sous la forme *si conditions alors actions* et sont utilisées dans de nombreux intergiciels, par exemple, FACTS [128], SOCAM [54], MidSen [101], REED [45] et LINC [79]. Les règles sont également utilisées dans plusieurs solutions de conception et/ou de déploiement de systèmes adaptatifs proposées dans la littérature. Certaines de ces solutions ciblent un domaine d'application spécifique (p. ex., bâtiment intelligent) tandis que les autres solutions proposent des approches génériques pour permettre la conception et/ou le déploiement de systèmes adapta-

tifs.

Les solutions proposées dans [86, 96, 83, 122, 74, 30] s'intéressent au domaine applicatif du bâtiment intelligent. Elles permettent de concevoir le système considéré sous la forme d'un ensemble de règles. Ces règles effectuent des actions lorsque des événements spécifiques surviennent, pour réaliser, par exemple, des objectifs de confort, de sécurité ou d'économie d'énergie. Par exemple, une règle peut être utilisée pour ouvrir la fenêtre d'une pièce lorsqu'une présence y est détectée et que le niveau de CO_2 est élevé, dans le but de ventiler la pièce. Une règle peut également être utilisée pour fermer la porte d'une pièce et éteindre toutes les lampes lorsque la pièce n'est pas occupée, pour des raisons de sécurité et d'économie d'énergie.

Les solutions proposées dans [49, 71, 66, 119, 14, 32] ne ciblent pas de domaines applicatifs particuliers. Dans ces solutions, la logique du système à adapter est mise en œuvre sous la forme d'un ensemble d'entités logicielles (p. ex., composants) et la logique d'adaptation est implémentée en utilisant des règles. Ces règles vérifient des conditions et effectuent des actions qui modifient les entités logicielles (p. ex., remplacer un composant par un autre) et/ou leurs interactions dans le but d'adapter le système. Ces solutions séparent la logique d'adaptation du système à adapter. Cette séparation permet la réutilisation de la logique d'adaptation et son évolution.

De plus, des études effectuées, dans le contexte du bâtiment intelligent, ont montré que les règles sont intuitives pour la description de comportements adaptatifs [37, 129]. L'ensemble des règles utilisées pour l'adaptation d'un système ne doit pas contenir des erreurs de conception, pour garantir la fiabilité comportementale.

Fiabilité comportementale des systèmes L'ensemble de règles utilisées pour l'adaptation d'un système peut contenir différents types d'erreurs :

- **conflit** : un conflit survient lorsque des règles différentes sont activées au même instant et ont des actions contradictoires sur le système considéré ;
- **violation d'objectifs** : une violation d'objectifs survient lorsque l'exécution d'une règle mène à un état, indésirable, violant un ou plusieurs objectifs ;
- **règle non activable** : une règle est non activable lorsque deux ou plusieurs conditions qu'elle vérifie sont incompatibles (p. ex., sont contradictoires) ;
- **incomplétude** : elle est caractérisée par le fait qu'il manque des règles ;
- **redondance** : elle survient lorsque des règles effectuent une même action ;
- **circularité** : elle correspond à une exécution en boucle de plusieurs règles.

La circularité peut correspondre au comportement d'un fonctionnement répétitif. Dans ce cas, elle n'est pas considérée comme une erreur. De même, la redondance peut être volontaire pour de la tolérance aux pannes. Lorsqu'elle est involontaire, la redondance peut causer des dégradations de performance (exécution de plusieurs règles) ou mener à des exécutions multiples d'une action qui ne doit pas être exécutée plusieurs fois (p. ex., injection d'une dose d'insuline à un patient) et est une erreur.

Certaines erreurs (conflit, violation d'objectifs, circularité et redondance) peuvent être explicites ou implicites. Elles sont implicites lorsqu'elles sont dues aux effets qui résultent de l'exécution des règles. Par exemple, une règle qui ouvre la fenêtre pour

ventiler une pièce peut violer, de façon implicite, un objectif qui limite le niveau de bruit de la pièce à un certain seuil. De même, une règle qui migre une entité logicielle d'une ressource de calcul à une autre peut violer un objectif qui stipule que deux entités logicielles utilisées pour de la redondance ne doivent pas être sur la même ressource de calcul. Enfin, une règle qui ferme toutes les ouvertures, d'une pièce, et une règle qui ferme toutes les portes et les fenêtres sont redondantes.

Plusieurs méthodes ont été proposées, dans la littérature, pour garantir la fiabilité comportementale des systèmes adaptatifs à base de règles. L'objectif est de détecter et de résoudre les erreurs qui peuvent être contenues dans un ensemble de règles. Pour détecter les erreurs, ces méthodes utilisent des mécanismes tels que la comparaison par paire de règles et la vérification formelle [12]. La vérification formelle consiste à modéliser la logique du système considéré en utilisant un langage formel (p. ex., réseaux de Petri [94], automates [59]) et à vérifier à l'aide d'un outil de vérification si un ensemble de propriétés, exprimées en logique temporelle, sont satisfaites par le modèle. Des exemples de telles propriétés sont l'absence de conflits et de violations d'objectifs, de redondance, de circularité et de règles non activables.

Le Tableau 2.1 présente des méthodes de détection et de résolution d'erreurs dans un ensemble de règles. Il spécifie pour chaque méthode les types d'erreurs qu'elle permet de détecter. Comme illustré par le tableau, les conflits explicites sont détectés par la plupart des méthodes alors que les erreurs implicites ne sont souvent pas détectées. La raison est que plusieurs méthodes de détection et de résolution d'erreurs ne considèrent pas les effets qui résultent de l'exécution des règles. Les méthodes proposées dans [76, 87, 116, 108], considèrent les effets de l'exécution des règles et permettent de détecter des conflits implicites et/ou des violations d'objectifs implicites. Dans [76, 87, 108], les conflits et/ou les violations d'objectifs détectés doivent être résolus de façon manuelle. Dans [116], les conflits détectés, au moment de l'exécution, sont résolues de façon automatique, en utilisant des règles de résolution qui sont prédéfinies, et une méthode est proposée pour la fiabilité d'exécution.

Fiabilité d'exécution des systèmes Plusieurs méthodes ont été proposées pour garantir la fiabilité d'exécution des systèmes adaptatifs à bases de règles [116, 107, 108, 39, 79]. Dans [116, 107, 108], les auteurs proposent une approche qui permet de détecter et d'éviter les inconsistances. Cette approche consiste à vérifier l'effet de l'exécution des règles en utilisant les données des capteurs ou d'autres sources. Par exemple, le capteur de luminosité peut être utilisé pour vérifier l'effet de la règle qui allume la lampe de la pièce. Lorsque l'action d'une règle n'est pas effectuée, une inconsistance est détectée. Par exemple, dans le cas de la règle qui allume la lampe d'une pièce, une inconsistance est détectée lorsque la luminosité de la pièce n'est pas égale à une valeur spécifique après l'exécution de la règle. Cependant, cette approche peut mettre du temps avant de détecter les inconsistances, dans le cas des actions dont les effets ne sont pas instantanés (p. ex., allumer le chauffage ou le climatiseur).

Dans [39, 79], les auteurs proposent d'utiliser des transactions distribuées [15]

TABLE 2.1 – Solutions de détection d'erreurs entre des règles

	COE	COI	VIE	VII	REN	INC	REE	REI	CIE	CII
Agusto, 2004 [8]	✔	✖	✖	✖	✔	✖	✔	✖	✔	✖
Cano, 2014 [22]	✔	✖	✔	✔	✔	✖	✔	✖	✔	✖
Khakpour, 2010 [67]	✔	✖	✔	✖	✔	✖	✔	✖	✖	✖
Le Guilly, 2016 [72]	✖	✖	✔	✖	✖	✖	✖	✖	✖	✖
Liang, 2015 [76]	✔	✔	✔	✔	✖	✖	✖	✖	✖	✖
Liu, 2014 [78]	✔	✖	✖	✖	✔	✖	✔	✖	✖	✖
Magill, 2016 [82]	✔	✖	✖	✖	✖	✖	✖	✖	✔	✖
Maternaghan, 2013 [87]	✔	✔	✖	✖	✖	✖	✖	✖	✖	✖
Nacci, 2015 [96]	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Preuveneers, 2016 [108]	✔	✔	✖	✖	✖	✖	✖	✖	✖	✖
Stavropoulos, 2015 [123]	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖
Shankar, 2005 [116]	✔	✔	✖	✖	✖	✖	✖	✖	✖	✖
Sun, 2015 [124]	✔	✖	✖	✖	✖	✖	✔	✖	✔	✖
Vannucchi, 2017 [131]	✖	✖	✔	✖	✔	✖	✔	✖	✖	✖

COE = conflit explicite, COI = conflit implicite, VIE = violation d'objectif explicite
VII = violation d'objectif implicite, REN = règle non activable, INC = incomplétude
REE = redondance explicite, REI = redondance implicite, CIE = circularité explicite
CII = circularité implicite

pour éviter les inconsistances. Une transaction permet d'exécuter un ensemble d'opérations de façon atomique. Elle est utilisée dans une règle pour effectuer des actions et mettre à jour les états logiques des actionneurs (états stockés dans le système). Lorsqu'une action ne peut pas être effectuée, par exemple, à cause d'une erreur de communication, l'état logique de l'actionneur correspondant n'est pas mis à jour et l'action n'est pas supposée être effectuée. Cela permet d'éviter les inconsistances au moment où les actions sont effectuées. Par exemple, le fait que le chauffage est en panne est détecté au moment de l'allumer et non après la baisse de la température de la pièce. Cependant, un actionneur peut tomber en panne après qu'une action ait été effectuée et créer une inconsistance. Par exemple, le chauffage peut tomber en panne après avoir été allumé. Dans ce cas, l'état logique du chauffage est égal à *allumé* alors qu'il ne l'est pas. Une telle inconsistance peut être détectée en utilisant les données des capteurs comme l'approche proposée dans [116, 107, 108]. Cette approche a été également proposée par [113, 114] pour détecter les inconsistances, dans le contexte du bâtiment intelligent, des processus métiers (« *business processes* »).

Conclusion L'adaptation basée sur des règles permet la conception intuitive de systèmes adaptatifs. De plus, de nombreuses méthodes ont été proposées pour garantir la fiabilité comportementale et la fiabilité d'exécution des systèmes conçus. Cependant, lorsque le système considéré est constitué de nombreuses entités, l'écriture des règles peut être fastidieuse. En effet, le développeur doit considérer manuellement tous les cas possibles. L'objectif est d'éviter que le système se retrouve dans des états dans lesquels aucune action ne peut être effectuée parce la règle correspondante n' a pas été définie. Pour éviter aux développeurs le fait de considérer l'ensemble des cas, plusieurs solutions de conception et/ou de déploiement de systèmes adaptatifs, proposées dans la littérature, se basent sur une fonction d'utilité.

2.2.3.2 Adaptation basée sur une fonction d'utilité

Elle consiste à adapter le système considéré de façon automatique en utilisant une fonction d'utilité. Une fonction d'utilité est une somme pondérée des propriétés du système (p. ex., qualité de service fournie, quantité d'énergie consommée). Les pondérations permettent de spécifier une préférence entre les propriétés. Par exemple, la qualité de service peut être préférée à la réduction de la consommation d'énergie. L'adaptation consiste à choisir, de façon automatique, la configuration du système qui offre la plus grande utilité lorsqu'un événement spécifique survient. Les fonctions d'utilité sont utilisées par des solutions de conception de systèmes adaptatifs (p. ex., [65, 89]) et par des intergiciels tels que MADAM [50] et CAMPUS [134].

MADAM et CAMPUS permettent d'adapter un système qui est développé sous la forme d'un ensemble d'entités logicielles (p. ex., composants). Pour utiliser ces intergiciels, le développeur décrit d'abord le système considéré (les entités logicielles et leurs interactions). Pour chaque entité logicielle, le développeur spécifie les différentes implantations et leurs propriétés. Une propriété décrit une qualité de service fournie (p. ex., temps de réponse) ou une ressource requise (p. ex., quantité de mémoire) et peut dépendre du contexte. Par exemple, le temps de réponse d'une entité peut dépendre de la bande passante disponible. Ensuite, le développeur définit, dans le cas de MADAM, la fonction d'utilité qui sera utilisée pour l'adaptation du système. Dans le cas de CAMPUS, une formule de fonction d'utilité prédéfinie permet de calculer et d'associer une valeur d'utilité à chaque entité logicielle. Dans les deux intergiciels, un gestionnaire d'adaptation est chargé de la reconfiguration automatique du système, sur la base d'une connaissance des entités logicielles et de leur propriétés. Lorsqu'un événement spécifique survient, le gestionnaire d'adaptation reconfigure le système en choisissant la configuration qui offre la plus grande utilité. Le même principe (choix d'une configuration qui offre la plus grande utilité) est utilisé par les solutions proposées dans [21, 92]. De plus, ces solutions proposent une méthode pour garantir la fiabilité comportementale des systèmes adaptatifs conçus.

Fiabilité comportementale des systèmes Pour garantir la fiabilité comportementale, certaines solutions, de conception et de déploiement de systèmes adaptatifs, basées sur des fonctions d'utilité, utilisent un outil de vérification formelle. L'objectif est de choisir, lorsqu'un événement spécifique survient, la configuration du système qui ne viole pas les objectifs considérés et qui offre la plus grande utilité. Un exemple d'une telle de solution est proposée dans [21]. Cette solution est basée sur les principes du calcul autonomique et utilise un gestionnaire autonomique pour effectuer l'adaptation du système considéré. Le gestionnaire autonomique possède comme connaissance sur le système à adapter un modèle de Markov [17], qui décrit le comportement du système. Ce modèle contient des paramètres qui sont mis à jour par l'observation du système (p. ex., temps d'exécution estimé d'un service) et est utilisé, par le gestionnaire autonomique, pour effectuer les opérations d'analyse et de planification. Ces opérations sont effectuées à l'aide de l'outil de vérification PRISM [70]. Cet outil permet d'explorer toutes les configurations possibles du sys-

tème, dans le but de déterminer celle qui offre la plus grande utilité, de façon fiable.

Fiabilité d'exécution des systèmes Les solutions de conception et de déploiement de systèmes adaptatifs basées sur des fonctions d'utilité se concentrent sur la prise de décisions. Elles proposent des approches pour choisir la configuration du système lorsqu'un événement spécifique survient et, dans la plupart des cas, ne spécifient pas comment la reconfiguration est mise en œuvre et comment la fiabilité d'exécution est garantie. Toutefois, les méthodes de fiabilité d'exécution proposées pour les système adaptatifs à base de règles peuvent être appliquées. Par exemple, les données des capteurs peuvent être utilisées pour vérifier l'exécution des actions.

Conclusion L'adaptation basée sur une fonction d'utilité permet l'automatisation de la prise des décisions d'adaptation. Cela permet aux développeurs de ne pas considérer tous les cas possibles pour permettre l'adaptation d'un système. Cependant, ils doivent définir des propriétés pertinentes du système à adapter et, dans la plupart des cas une fonction d'utilité, ce qui n'est pas une tâche facile [69]. De plus, l'utilisation en ligne d'un outil de vérification qui permet de garantir la fiabilité comportementale des systèmes conçus peut causer des dégradations de performance.

2.2.3.3 Adaptation basée sur la théorie du contrôle

Elle consiste à spécifier, dans un premier temps, les objectifs à réaliser et à définir un modèle du système à adapter. Ensuite, à concevoir un contrôleur qui, à partir du modèle et des objectifs, adapte le système aux changements de son environnement. L'adaptation basée sur la théorie du contrôle est utilisée par plusieurs solutions pour concevoir des systèmes adaptatifs [102, 117]. Des exemples de telles solutions sont celles proposées dans [118] et [68]. Dans [118], les auteurs proposent une approche générique pour la conception de systèmes adaptatifs dans lesquels plusieurs objectifs doivent être réalisés. Dans [68], les auteurs s'intéressent aux bâtiments intelligents et proposent une approche réalisant des objectifs de confort et d'économie d'énergie. Ces solutions sont basées sur la théorie du contrôle continu ou du contrôle discret.

Contrôle continu Le modèle du système est fourni sous une forme mathématique (p. ex., des équations différentielles) qui décrit la dynamique du système. Les objectifs constituent la consigne de contrôle qui permet au contrôleur d'adapter le système. Pour effectuer l'adaptation, comme illustré à la Figure 2.2, le contrôleur calcule, en fonction de l'écart entre la sortie du système et la consigne (l'erreur), une commande qui est appliquée au système. Cette commande permet d'adapter le système de sorte que la sortie soit égale à la consigne (ou à la plus proche valeur possible) et ceci malgré les perturbations extérieures (p. ex., variation de charges).

Le contrôleur peut être conçu sous plusieurs formes. Il peut être, par exemple, un PI (Proportionnel Intégral), un PID (Proportionnel Intégral Dérivé) ou basé sur une technique de contrôle avancée (p. ex., commande prédictive, commande H_∞). Le PID est, d'après [117], le contrôleur le plus utilisé dans la littérature pour la

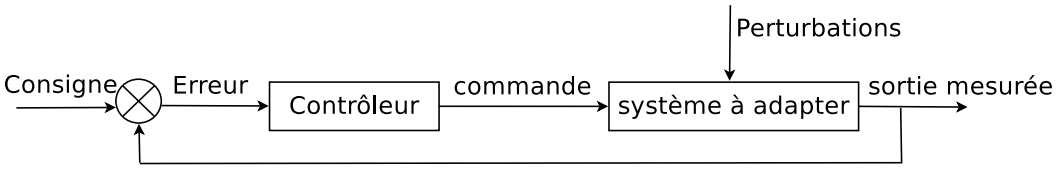

FIGURE 2.2 – Boucle de contrôle continu

conception de systèmes adaptatifs. La raison est qu'il permet d'obtenir une erreur nulle (la sortie du système est égale à la consigne) malgré les perturbations et est facile à concevoir. La commande (u) à appliquer sur le système est calculée en fonction de l'erreur (e), entre la consigne et la sortie mesurée, en utilisant la formule suivante :

$$u(t) = K(e(t) + \frac{1}{T_i}\int_0^t e(\tau)\,\mathrm{d}\tau + T_d\frac{\mathrm{d}e(t)}{\mathrm{d}t}) \tag{2.1}$$

où K, T_i, T_d sont des gains à définir. Ces gains peuvent être définis par le calcul en utilisant le modèle du système ou de façon empirique, en utilisant des méthodes telles que celle de Ziegler et Nichols [130]. Une fois les gains définis, le PID peut être programmé dans un langage (p. ex., python) et utilisé pour adapter le système. Cependant, le PID ne permet pas de contrôler des systèmes qui possèdent plusieurs entrées et plusieurs sorties. Pour l'adaptation de tels systèmes, la technique de contrôle la plus utilisée dans la littérature est basée sur la commande prédictive [117]. Enfin, pour les systèmes dont l'adaptation consiste à prendre des décisions logiques (p. ex., démarrer ou arrêter une ressource de calcul), les solutions de systèmes adaptatifs, proposées dans la littérature, utilisent le contrôle discret.

Contrôle discret Le modèle du système considéré est d'abord fourni sous la forme d'un système de transitions, (cf. Figure 2.3), qui peut être par exemple un automate [59] ou un réseau de Petri [94]. Ensuite, le système de transitions conçu est utilisé pour effectuer la vérification formelle, comme dans certaines méthodes proposées, dans la littérature, pour la détection d'erreurs dans un ensemble de règles.

La vérification formelle [12] consiste à vérifier, à l'aide d'un outil de vérification, si un ensemble de propriétés, exprimées en logique temporelle, sont satisfaites par le système de transitions qui modélise le système considéré. Des exemples de telles propriétés sont : le fait qu'une action soit toujours effectuée après un événement spécifique ou le fait que deux actions ne puissent pas être effectuées au même instant. La vérification formelle permet de garantir la fiabilité comportementale des systèmes et est utilisée, par exemple, dans [55, 5, 10, 29, 103, 9]. Cependant, elle requiert de programmer manuellement, en utilisant un système de transitions, la logique du système (les différentes entités du système, leurs comportements et leurs interactions pour réaliser les objectifs). De plus, lorsqu'une propriété n'est pas satisfaite, la logique programmée du système doit être modifiée et le système de transitions obtenu doit être vérifié à nouveau. Du fait de la programmation manuelle de la logique du

système et de sa modification, la vérification formelle peut devenir fastidieuse.

Pour surmonter cette limitation, certaines solutions de conception et/ou de déploiement de systèmes adaptatifs (p. ex., [3, 56, 137, 56]), se basent sur la synthèse de contrôleurs discrets [59]. Dans ce cas, le système de transitions modélise les entités du système et leurs comportements : les comportements désirables et les comportements indésirables par rapport aux objectifs du système et est utilisé par un outil de synthèse de contrôleurs. Le but est de contrôler le système de transitions de sorte à n'autoriser que les comportements qui ne violent pas les objectifs considérés.

Pour ce faire, les variables d'entrées du système de transitions doivent être divisées en deux catégories : variables non contrôlables et variables contrôlables. Les variables non contrôlables sont celles dont les valeurs ne dépendent pas du système (p. ex., la panne d'une ressource de calcul, la présence d'une personne dans une pièce). Les variables contrôlables sont utilisées pour effectuer la synthèse du contrôleur, par un outil de synthèse de contrôleurs. L'outil de synthèse de contrôleurs explore, en mode hors ligne, l'espace d'états du système de transitions et calcule les valeurs possibles des variables contrôlables dans le but de réaliser les objectifs qu'elles que soient les valeurs des variables non contrôlables. Le résultat de la synthèse est un contrôleur qui, comme illustré à la Figure 2.4, donne des valeurs aux variables contrôlables en fonction des valeurs des variables non contrôlables et de l'état courant du système de transitions. Cela déclenche une transition qui adapte le système de transitions à son environnement tout en réalisant les objectifs considérés.

Le système de transitions contrôlé doit être relié au système réel à l'aide de capteurs et d'actionneurs. Le but est de collecter les entrées et d'exécuter les sorties pour adapter le système et garantir sa fiabilité comportementale. L'exécution doit être effectuée, de façon fiable, en prenant en compte le fait que les actionneurs peuvent tomber en panne ou devenir inaccessibles à cause d'une erreur de communication.

Fiabilité comportementale des systèmes La théorie du contrôle, grâce à ses fondements mathématiques, garantit la fiabilité comportementale des systèmes adaptatifs conçus. Dans le cas du contrôle continu, la commande calculée par le contrôleur permet d'atteindre la consigne de contrôle malgré les perturbations extérieures. Dans le cas de la vérification formelle, l'outil de vérification permet de garantir que les décisions d'adaptation ne sont pas conflictuelles et ne violent pas des objectifs du système considéré. Enfin, dans le cas de la synthèse de contrôleurs discrets, le contrôleur généré calcule des commandes qui sont à la fois correctes et cohérentes.

Fiabilité d'exécution des systèmes Dans [56, 51, 40], les auteurs proposent une approche qui consiste à modéliser dans le système de transitions le fait que certaines entités du système peuvent tomber en panne. Dans ce cas, des variables non contrôlables sont associées à ces entités pour spécifier si elles sont ou pas en panne. Ces variables sont des entrées du contrôleur et doivent être collectées à chaque instant (les pannes des entités doivent être détectées), par exemple en utilisant les données des capteurs. En fonction des valeurs des variables non contrôlables, le

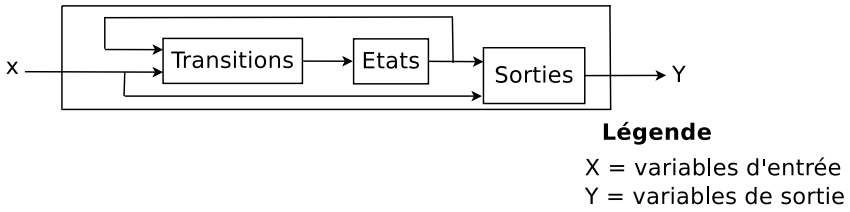

FIGURE 2.3 – Système de transitions

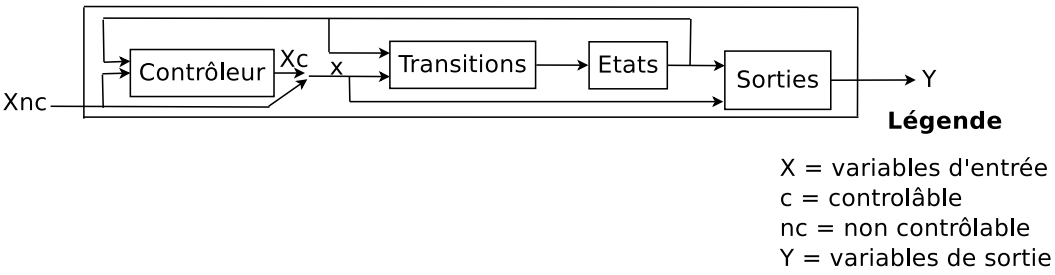

FIGURE 2.4 – Système de transitions contrôlé

contrôleur décide des actions à effectuer pour adapter le système. Cette approche permet d'éviter les inconsistances causées par des pannes matérielles. En effet, cette approche oblige à détecter les pannes pour les fournir en entrée du contrôleur. De plus, elle permet, lorsqu'une panne est détectée, d'effectuer une action alternative.

Dans des solutions telles que celles proposées dans [55, 60], les auteurs effectuent la vérification formelle et modélisent les pannes qui peuvent survenir. Cela permet de vérifier que le système se comporte correctement même en présence de pannes.

Conclusion La théorie du contrôle permet l'adaptation des systèmes et garantit, du fait des ses fondements mathématiques, leur fiabilité comportementale. De plus, la théorie du contrôle continu et la synthèse de contrôleurs discrets permettent l'automatisation de la prise des décisions d'adaptation. En effet, les actions à effectuer pour adapter le système considéré sont décidées de façon automatique par un contrôleur, comme illustré dans [56, 137, 68], dans domaine du bâtiment intelligent.

2.2.4 Solutions spécifiques au bâtiment intelligent

Dans la littérature, de nombreuses solutions ont été proposées pour permettre la conception et le déploiement de systèmes de gestion de bâtiments intelligents. Certaines de ces solutions proposent des interfaces graphiques de programmation qui sont destinées aux occupants ou aux gestionnaires des bâtiments. Des solutions ont été également proposées pour éviter les décisions conflictuelles implicites en spécifiant les effets que les actions effectuées sur les actionneurs ont sur l'environnement.

2.2.4.1 Interfaces graphiques de programmation

Plusieurs interfaces graphiques de programmation ont été proposées dans le contexte du bâtiment intelligent [48, 73, 115, 80, 121, 68]. L'objectif est de permettre aux occupants, ou aux gestionnaires des bâtiments, de contrôler leurs bâtiments en définissant des règles. Les occupants ou les gestionnaires de bâtiments spécifient les actions à effectuer lorsque des événements spécifiques surviennent. Par exemple, pour éclairer une pièce, un occupant peut écrire deux règles : une règle qui ouvre le volet pour utiliser la lumière du jour et une autre règle qui allume la lampe lorsque la lumière du jour n'est plus disponible. L'utilisation de ces interfaces ne requiert pas de connaissances en informatique. Cependant, comme ces interfaces sont basées sur des règles, leur utilisation pour contrôler un bâtiment requiert la considération de tous les cas possibles. Par exemple, pour éclairer une pièce équipée d'un volet et d'une lampe, plusieurs règles doivent être définies pour spécifier quand est ce que le volet doit être ouvert, fermé, et quand est ce que la lampe doit être allumée, éteinte.

2.2.4.2 Effets des actionneurs sur l'environnement

Plusieurs solutions proposées pour la fiabilité des systèmes de gestion de bâtiments intelligents ont souligné la nécessité de considérer les effets des actionneurs sur l'environnement. La raison est que ces effets peuvent être à l'origine de décisions conflictuelles ou de violations d'objectifs qui sont implicites et difficile à détecter. Des exemples de telles solutions sont celles proposées dans [93, 87, 97]. Dans [93], les auteurs considèrent d'abord, onze paramètres de l'environnement (p. ex., température, humidité, luminosité, odeur). Ensuite, ils définissent des opérateurs qui permettent de spécifier les effets des actionneurs sur ces paramètres. Ces opérateurs sont *augmente*, *diminue* et *change*. Par exemple, un chauffage augmente la température. De même, une lampe augmente la luminosité. Dans [87, 97], les auteurs spécifient les effets des actionneurs de façon plus précise. Ils proposent de spécifier de combien un actionneur augmente, diminue ou change un paramètre de l'environnement. Par exemple, le fait d'allumer une lampe augmente la luminosité de 200 lux. Le fait de spécifier de combien un actionneur affecte un paramètre de l'environnement permet de détecter plus de conflits et de violations d'objectifs. Par exemple, cela permet de détecter que le fait d'allumer, dans une pièce, une seule lampe qui fournit 200 lux viole un objectif qui consiste à fournir une luminosité supérieure ou égale à 500 lux.

2.2.4.3 Synthèse des solutions pour les bâtiments intelligents

Dans le contexte du bâtiment intelligent, plusieurs interfaces graphiques ont été proposées pour permettre aux occupants ou aux gestionnaires des bâtiments de contrôler le comportement des actionneurs. De plus, certaines solutions proposées pour garantir la fiabilité comportementale des systèmes de gestion de bâtiments considèrent les effets des actionneurs sur l'environnement. Ces solutions permettent de détecter les décisions qui, de façon implicite, sont conflictuelles ou violent les

objectifs à réaliser. Cependant, dans la plupart de ces solutions, les violations d'objectifs et les conflits implicites détectés doivent être résolus de façon manuelle.

2.2.5 Conclusion

Plusieurs solutions ont été proposées, dans la littérature, pour la conception et le déploiement de systèmes adaptatifs. Ces solutions sont basées sur des règles, des fonctions d'utilité ou des objectifs pouvant être réalisés avec la théorie du contrôle.

Les solutions basées sur des règles sont intuitives pour la conception de systèmes adaptatifs. Cependant, elles peuvent devenir fastidieuses lorsque le système considéré est constitué d'un nombre élevé d'entités. Dans ce cas, le développeur doit considérer tous les cas possibles pour décrire les actions à effectuer lorsque des événements spécifiques surviennent. Dans les solutions basées sur des fonctions d'utilité, les actions à effectuer sont décidées par un gestionnaire d'adaptation, sur la base d'une connaissance du système et une fonction d'utilité fournies par le développeur. Enfin, dans le cas des solutions basées sur la théorie du contrôle, les actions à effectuer sont décidées par un contrôleur. Ce contrôleur garantit, grâce aux fondements mathématiques de la théorie du contrôle, la fiabilité comportementale du système.

Du fait de la nature distribuée des systèmes adaptatifs, plusieurs solutions proposées pour leur conception et leur déploiement utilisent un intergiciel. Certaines solutions garantissent la fiabilité comportementale et/ou d'exécution des systèmes conçus. Enfin, pour permettre l'adaptation autonome des systèmes, plusieurs solutions proposées dans la littérature se basent sur les principes du calcul autonomique.

Le Tableau 2.2 compare plusieurs solutions proposées, dans la littérature, pour la conception, le déploiement et la fiabilité des systèmes adaptatifs. Ce tableau présente pour chaque solution la technique d'adaptation employée et spécifie si elle utilise un intergiciel. Le Tableau précise également pour chaque solution si elle garantit la fiabilité comportementale et la fiabilité d'exécution des systèmes adaptatifs conçus.

Comme illustré dans le Tableau 2.2, la fiabilité comportementale et la fiabilité d'exécution ne sont pas à la fois considérées par beaucoup de solutions. Les solutions (p. ex., [116, 108, 56]) qui combinent ces deux forme de fiabilité sont basées sur des règles ou n'utilisent pas d'intergiciels. Les solutions basées sur des règles requièrent de considérer tous les cas pour décrire les actions à effectuer. Celles n'utilisant pas d'intergiciels requièrent la gestion des interactions entre les entités du système considéré. De plus, ces solutions, combinant les deux formes de fiabilité, se basent sur les données des capteurs pour détecter les inconsistances. Une inconsistance est détectée après l'exécution de l'action correspondante et la détection peut prendre du temps, dans le cas d'une action dont les effets ne sont pas instantanés. Par exemple, considérons une pièce équipé d'un chauffage, qui est en panne, et une action qui consiste à allumer le chauffage. Lorsque les données des capteurs sont utilisées, le fait que l'action n'est pas effectuée est détecté, lorsque la température n'a pas changé, après un certain temps. Ce qui crée une inconsistance entre l'instant où l'action est effectuée et l'instant où la panne est détectée, pouvant être un problème. Une telle inconsistance peut être évitée en utilisant des transactions distribuées comme

TABLE 2.2 – Solutions de conception et de déploiement de systèmes adaptatifs

	Technique d'adaptation	Intergiciel	Fiabilité	
			comportementale	exécution
SOCAM [54]	Règles	✔	✖	✖
DACAR [38]	Règles	✔	✔	✖
PobSAM [66]	Règles	✖	✔	✖
ECA-P [116]	Règles	✖	✔	✔
BuildingRules [96]	Règles	✖	✔	✖
SIFT [76]	Règles	✖	✔	✖
[108]	Règles	✔	✔	✔
MADAM [50]	Fonction d'utilité	✔	✔	✖
CAMPUS [134]	Fonction d'utilité	✔	✔	✖
QosMOS [21]	Fonction d'utilité	✖	✔	✖
SimCA [118]	Contrôle continu	✖	✔	✖
Ctrl-F [3]	Contrôle discret	✔	✔	✖
[56]	Contrôle discret	✖	✔	✔
[68]	Contrôle continu	✖	✔	✖

dans [39, 79]. Dans ce cas, le fait qu'une action ne peut pas être effectuée parce que l'actionneur correspondant est en panne est détecté au moment d'effectuer l'action.

Pour résoudre ces différentes limitations, cette thèse propose un support intergiciel pour la conception et le déploiement de systèmes adaptatifs, garantissant à la fois la fiabilité comportementale et la fiabilité d'exécution des systèmes conçus. Pour ce faire, ce support intergiciel repose sur la théorie du contrôle, les transactions distribuées et la vérification des actions effectuées, à l'aide des capteurs, pour la détection de pannes. De plus, le support intergiciel proposé est basé sur les principes du calcul autonomique pour permettre l'adaptation autonome des systèmes. Ce support intergiciel permet de générer des modèles comportementaux et des modèles d'exécution pour les systèmes et est mis en œuvre en utilisant différents outils.

2.3 Outils utilisés

Le support intergiciel proposé, dans cette thèse, pour la conception et le déploiement de systèmes adaptatifs fiables est mis en œuvre à l'aide de trois outils. Ces outils sont : un intergiciel à base de tuples qui supporte les transactions distribuées (LINC [79]), un environnement d'abstraction (PUTUTU [100]) et un langage réactif permettant d'effectuer de la synthèse de contrôleurs discrets (Heptagon/BZR [36]).

L'intergiciel à base de tuples supportant les transactions permet de bénéficier du découplage spatial et temporel des différentes entités des systèmes et d'une interface de programmation applicative simple pour la gestion de leurs interactions. De plus, il permet, en partie grâce aux transactions, de garantir la fiabilité d'exécution des systèmes. L'environnement d'abstraction permet de gérer l'hétérogénéité des capteurs et des actionneurs qui constituent les systèmes. Enfin, le langage réactif supportant la synthèse de contrôleurs discrets permet de garantir la fiabilité comportementale des systèmes. Ces outils (LINC, PUTUTU et H/BZR) sont présentés par la suite.

2.3.1 LINC

LINC [79] est un intergiciel, à base de tuples, utilisé pour concevoir et déployer des applications distribuées. Il fournit un langage à base de règles pour la conception des applications distribuées et des mécanismes pour permettre leur déploiement.

2.3.1.1 Langage LINC

LINC fournit un langage qui permet de concevoir une application distribuée sous la forme d'un ensemble d'objets et de règles. Ce langage repose sur trois paradigmes :

- **mémoire associative** [24] : elle consiste à modéliser l'application considérée sous la forme de sacs contenant des tuples. Les sacs sont regroupés, selon la logique de l'application, dans des entités logicielles appelées objets. Les tuples sont manipulées à l'aide de trois opérations : *rd*, *get* et *put*. L'opération *rd* permet de vérifier la présence d'un tuple dans un sac. Les opérations *get* et *put* permettent, respectivement, de consommer et d'ajouter des tuples dans des sacs. Ces trois opérations sont utilisées dans des règles de production ;
- **règles de production** [28] : une règle de production est constituée de deux parties : une *précondition* et une *performance*. Dans la *précondition*, l'opération *rd* est utilisée, avec comme paramètre un tuple partiellement instancié, pour vérifier des conditions sur le système. Lorsque les conditions sont vraies, la *performance* est déclenchée. Dans la *performance* les trois opérations *rd*, *get* et *put* sont utilisées. Les tuples manipulés dans la *performance* d'une règle doivent être complètement instanciés. Le *rd* est utilisé pour vérifier des conditions. Le *get* et le *put* sont utilisés pour effectuer des actions sur le système et mettre à jour son état logique (tuples stockés dans des sacs) ;
- **transactions distribuées** [15] : elles sont utilisées dans la *performance* d'une règle LINC. Une transaction permet de regrouper en une seule opération : la vérification des conditions (*rd*), la réalisation des actions (*put*) et la mise à jour de l'état logique du système (*get*, *put*). Ainsi, la *performance* d'une règle peut être annulée si, par exemple, la vérification d'une condition à l'aide d'une opération *rd* n'est plus vraie. La *performance* s'arrête également si une opération *put* échoue parce que l'action correspondante (p. ex., allumer une lampe) ne peut pas être effectuée (p. ex, du fait d'une panne).

Exemple de règle LINC Considérons une pièce contenant trois ressources de calcul qui sont allumées. Le Listing 2.1 présente une règle LINC qui éteint toutes les ressources de calcul cette pièce lorsqu'une présence n' y est pas détectée.

La *précondition* de la règle (avant le symbole `::`) vérifie d'abord si le capteur de présence, dont l'identifiant est égal à `pres_12`, n'a pas détecté une présence. Pour ce faire, elle effectue une opération *rd* sur le sac `Sensors` de l'objet `ModBus` (technologie de communication du capteur de présence). Ensuite, la *précondition* vérifie s'il y a, dans la pièce, des ressources de calcul qui sont allumées, en effectuant une opération *rd* sur le sac `Etat` de l'objet `RscCalcul`. Ce sac associe l'identifiant d'une ressource

```
1 ["ModBus","Sensors"].rd("pres_12","false")&
  ["RscCalcul","Etat"].rd(id,"allumee")
3 ::
  {
5  ["ModBus","Sensors"].rd("pres_12","false");
   ["RscCalcul","Commande"].put(id,"eteindre");
7  ["RscCalcul","Etat"].get(id,"allumee");
   ["RscCalcul","Etat"].put(id,"eteinte")
9 }.
```

Listing 2.1 – Exemple de règle LINC

de calcul à son état logique. L'opération *rd* sur ce sac (ligne 4) retourne, un par un dans la variable `id`, les identifiants des ressources de calcul qui sont allumées. Pour chaque identifiant retourné, la *performance* de la règle est déclenchée et exécutée.

La *performance* de la règle est constituée d'une seule transaction (entre {}). Cette transaction vérifie d'abord le fait que la présence n'est toujours pas détectée dans la pièce (ligne 5). Ensuite, elle éteint la ressource de calcul dont l'identifiant est spécifié dans la variable `id` et change son état logique (lignes 6 et 8). LINC garantit que toutes les opérations d'une transaction sont effectuées ou aucune d'elles ne l'est. Par exemple, lorsqu'une ressource de calcul (p. ex., `rsc12`) ne peut pas être éteinte du fait d'une erreur de communication, l'opération *put* de la ligne 6 échoue et les autres opérations de la transaction ne sont pas exécutées pour `rsc12`. Dans ce cas, l'état logique de `rsc12` reste égal à `allumee` et est consistant avec son état réel.

Exécution des règles LINC Les règles, d'une application LINC, sont compilées et exécutées, en parallèle, par des objets LINC. Un objet peut exécuter plusieurs règles. Pour chaque règle, l'objet associé exécute d'abord la *précondition*. Pour ce faire, l'objet évalue chaque opération *rd* et construit un arbre d'inférence avec instanciation et la propagation des variables. Pour chaque branche dont la profondeur est égale au nombre d'opérations *rd* utilisées dans la *précondition*, la *performance* de la règle est déclenchée et est exécutée par l'objet. Plusieurs instances de la *performance* peuvent être déclenchées, en parallèle, pour différentes branches.

Dans la *performance*, les transactions d'une règle sont exécutées en séquence. Chaque transaction implante un protocole de validation exécutée en deux phases (« *two-phase commit protocol* »). Dans la première phase d'une transaction, des pré-opérations (*pre_rd*, *pre_get*, *pre_put*) sont exécutées pour voir si les opérations correspondantes peuvent être effectuées. Lorsqu'une pré-opération échoue, du fait d'une erreur de communication ou d'une panne, la première phase échoue et l'exécution de la transaction s'arrête. Lorsque l'exécution d'un *pre_rd* ou d'un *pre_get* est réussie, le tuple associé est verrouillé et ne peut plus être utilisé par une autre transaction. Une autre transaction voulant utiliser le même tuple attend jusqu'à ce qu'il soit libéré, à la fin de l'exécution de la transaction qui est en cours. Lorsque toutes les pré-opérations sont réussies, la deuxième phase de la transaction est exécutée.

La deuxième phase de la transaction consomme (*get*) ou libère (*rd*) les tuples

qui ont été verrouillés, lors de la première phase, et insère les tuples associés aux opérations *put*. L'exécution de la deuxième phase de la transaction est toujours réussie parce qu'il a été vérifié, dans la première phase, que toutes les opérations peuvent être effectuées. Cependant, des erreurs de communication ou des pannes peuvent survenir après les vérifications effectuées dans la première phase et font que certaines opérations ne peuvent plus être effectuées. De telles erreurs et pannes ne sont pas détectées, avant la fin de la transaction, parce qu'elles sont survenues après la phase de vérification. De telles erreurs de communication et pannes sont traitées comme si elles étaient survenues juste après que la transaction ait été effectuée.

Par exemple, considérons une ressource de calcul allumée, connectée au réseau local et une transaction T_1 utilisée pour l'éteindre. Dans la première phase, la transaction vérifie si la ressource de calcul peut être éteinte (elle n'est pas en panne ou inaccessible à cause d'une erreur de communication) par exemple en effectuant une commande *Ping*. Dans la deuxième phase, la transaction éteint la ressource de calcul puisqu'il a été vérifié, dans la première phase, qu'elle pouvait être éteinte. Cependant, La ressource de calcul peut tomber en panne ou devenir inaccessible juste après la vérification effectuée dans la première phase. Dans ce cas, comme la vérification a été déjà effectuée, la panne (ou l'erreur de communication) n'est pas détectée par la transaction T_1, elle sera détectée par une autre transaction qui voudra effectuer une action sur la ressource de calcul (p. ex., allumer). C'est comme si la panne était survenue juste après que la ressource de calcul ait été éteinte par T_1.

Exemple d'exécution d'une règle LINC Considérons la règle présentée au Listing 2.1. L'objet qui exécute cette règle commence par évaluer la première opération *rd* (ligne 1). Cela retourne, un par un, tous les tuples qui sont contenus dans le sac `Sensors` de l'objet `Modbus` qui correspondent au motif `(pres_12,"false")`. Si un tel tuple n'existe pas dans le sac `Sensors` (le capteur a détecté une présence), le *rd* reste bloqué et l'exécution de la règle s'arrête, jusqu'à ce que un tel tuple soit inséré. Lorsqu'un tel tuple existe (une présence n'est pas détectée), il est retourné, une branche de l'arbre d'inférence est créée et l'opération *rd* de la ligne 4 est évaluée. De même, cela retourne, un par un, tous les tuples du sac `States` de l'objet `RscCalcul` qui correspondent au motif `(id,"allumee")`. Pour chaque tuple retourné, la variable `id` est instanciée et une nouvelle branche est créée dans l'arbre d'inférence. Pour chaque branche de profondeur égale à deux (le nombre d'opérations *rd* utilisées dans la *précondition* de la règle), la *performance* de la règle est exécutée pour éteindre une ressource de calcul et mettre à jour son état logique.

2.3.1.2 Déploiement des applications LINC

Dans LINC, l'unité de déploiement est l'objet. Un objet contient un ou plusieurs sacs et possède un type. Le type d'un objet définit ses sacs et son code (modules Python). Le type peut être utilisé pour définir un nouveau type d'objet en effectuant de l'héritage. LINC fournit des mécanismes pour démarrer, arrêter et migrer des objets. Il fournit également des mécanismes pour activer et désactiver des règles.

Démarrage et arrêt d'un objet Un objet peut être démarré, ou arrêté, sur une ressource de calcul. Pour ce faire, un processus démon est exécuté sur la ressource de calcul. Le démon peut être lancé à distance (p. ex., en utilisant ssh) ou automatiquement lorsque la ressource de calcul s'allume. Le démon fournit une URL pour démarrer et arrêter des objets sur la ressource de calcul. Pour démarrer un objet, son code doit être présent sur la ressource de calcul. Pour ce faire, le code de l'objet peut être téléchargé à partir d'un dépôt ou copié d'une autre ressource de calcul.

Une fois démarré, un objet s'inscrit au *NameServer*. Le *NameServer* est un objet spécifique qui contient des informations (p. ex., emplacement) sur les autres objets dans une application LINC. Lorsqu'un objet A, exécutant une règle, veut communiquer avec un objet B (p. ex., lire un tuple dans un sac), il demande d'abord les informations pertinentes au *NameServer*. Ensuite, il crée, à partir des informations fournies par le *NameServer*, un stub pour communiquer directement avec l'objet B.

Migration d'un objet LINC permet de migrer, pour une application, un objet d'une ressource de calcul à une autre sans arrêter l'application. D'abord, l'objet est gelé. Pour ce faire, toutes les transactions, dans lesquelles l'objet est impliqué, sont terminées et aucune nouvelle transaction n'est acceptée. Ensuite, l'état interne de l'objet (le contenu de ses sacs) est sauvegardé dans un tuple. Ce tuple est ensuite déplacé et l'objet correspondant est re-instancié sur la nouvelle ressource de calcul. Au cours de la ré-instanciation, le *NameServer* est mis à jour avec les nouvelles informations. Durant le processus de migration, tous les objets, exécutant des règles, qui essaient de communiquer avec l'objet à migrer (M) recevront une erreur de communication. Ces objets vont attendre un certain temps et vont demander à nouveau, au *NameServer*, des informations pour communiquer avec l'objet M. À un instant donné, l'objet M sera démarré sur la nouvelle ressource de calcul, le *NameServer* sera mis à jour et les objets peuvent à nouveau communiquer avec l'objet M et exécuter leurs règles. Ce processus de migration est présenté en détail dans [6] où il est mis en œuvre avec un version précédente de l'intergiciel LINC.

Activation et désactivation d'une règle Lorsqu'un objet compile une règle, il génère d'abord un identifiant (p. ex., `Ru_001`). Ensuite, il ajoute une opération *rd* au début de la *précondition* et au début de chaque transaction de la règle, comme illustré au Listing 2.2. La variable `ego` est remplacée au moment de la compilation par le nom de l'objet exécutant la règle. Chaque objet possède un sac appelé *RulesId*. Ce sac associe l'identifiant de chaque règle, exécutée par l'objet, à un état qui peut être `ENABLED` (actif) ou `DISABLED` (inactif). Le sac *RulesId* d'un objet est utilisé pour activer ou désactiver des règles, en consommant et en insérant des tuples.

Par exemple, considérons la règle présentée au Listing 2.2. Pour désactiver cette règle, le tuple `("Ru_001","ENABLED")` est consommé du sac *RulesId* et le tuple `("Ru_001","DISABLED")` y est inséré. Dans ce cas, aucune nouvelle précondition n'est commencée et toutes les transactions échoueront lors de la première opération (ligne 5). En d'autres termes cela garantit que la règle cesse d'avoir un effet sur le sys-

```
1 [ego, "RulesId"].rd("RU_001","ENABLED") &
  ... #autres rd
3 ::
  {
5 [ego, "RulesId"].rd("RU_001","ENABLED");
   #autres operations
7 }.
```

Listing 2.2 – Opérations *rd* ajoutée lors de la compilation d'une règle

tème, même si ses événements déclencheurs se produisent. La réactivation de la règle est effectuée en remplaçant, dans le sac *RulesId*, le tuple `("Ru_001","DISABLED")` par `("Ru_001","ENABLED")`. Cela déclenchera une nouvelle *précondition*. Plus de détails sur l'activation et la désactivation des règles peuvent être trouvés dans [79].

2.3.2 PUTUTU

PUTUTU [100, 39] est un environnement d'abstraction fourni par l'intergiciel LINC. Il permet de communiquer avec des capteurs et des actionneurs et de masquer leur hétérogénéité. PUTUTU est constitué d'un ensemble d'objets LINC. Comme illustré à la Figure 2.5, ces objets encapsulent différentes technologies de communication (p. ex., `TelosB`, `EnOcean`, `Tellstick`) et héritent de quatre objets génériques :

- ***Object_dongles_modules*** : il est utilisé pour gérer un dongle ou tout autre équipement connecté à un port Ethernet ou un port USB. Cet objet permet de communiquer avec les capteurs et/ou les actionneurs d'une technologie spécifique et contient deux sacs : *Type* et *Location*. *Type* associe l'identifiant d'un capteur (d'un actionneur) à son type (p. ex., capteur de CO_2). *Location* associe l'identifiant d'un capteur (celui d'un actionneur) à son emplacement ;
- ***Object_wsan_sensors*** : il est utilisé pour communiquer avec des capteurs. Cet objet contient un sac supplémentaire appelé *Sensors*. Ce sac associe l'identifiant d'un capteur à la valeur qu'il a mesurée, sous la forme *(id,valeur)* ;
- ***Object_wsan_actuators*** : il est utilisé pour communiquer avec des actionneurs. Cet objet contient un sac supplémentaire appelé *Actuators* qui est utilisé pour envoyer des commandes aux actionneurs. Les tuples de ce sac sont sous la forme *(id,commande,paramètres)*. L'insertion d'un tel tuple, en utilisant l'opération *put*, envoie la commande à l'actionneur qui est spécifié ;
- ***Object_wsan_sensors_actuators*** : il est utilisé pour gérer les technologies fournissant à la fois des capteurs et des actionneurs (p. ex., `EnOcean`). Cet objet hérite des deux objets génériques précédents et contient leurs sacs.

2.3.3 Heptagon/BZR

Heptagon/BZR (H/BZR) [36] est un langage, flot de données, synchrone, utilisé pour la mise œuvre de systèmes réactifs. Il appartient à la même famille que les langages Lustre, Esterel et Signal [4]. Ces langages sont basés l'hypothèse synchrone [4] : une réaction du système est supposée être plus rapide que la dynamique

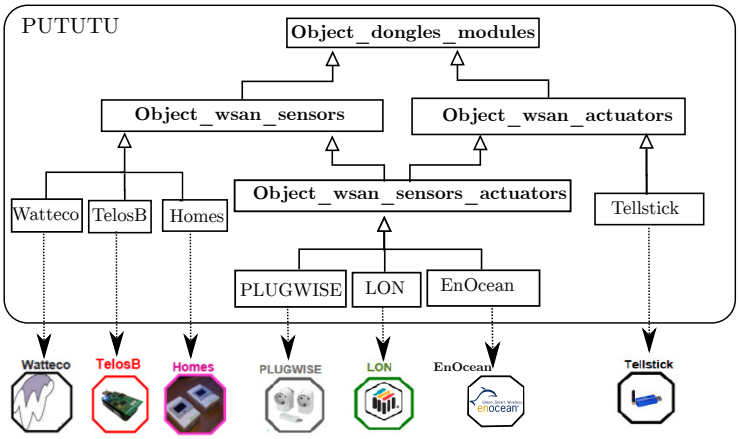

FIGURE 2.5 – Environnement d'abstraction PUTUTU

du système. Dans ces langages, le comportement réactif du système conçu peut être périodique ou basé sur des événements [4]. Dans le premier cas, une réaction est déclenchée périodiquement (p. ex., toutes les 5 secondes). Dans le deuxième cas, une réaction est déclenchée à chaque fois qu'un événement survient dans le système.

Dans H/BZR, chaque entité du système considéré peut être modélisée sous la forme d'un automate. Ensuite, les différents automates peuvent être composés, en parallèle ou en hiérarchie, pour obtenir un automate global qui représente le comportement du système considéré. La spécificité de H/BZR est qu'il permet d'effectuer, lors de la compilation des programmes conçus, de la synthèse de contrôleurs discrets.

2.3.3.1 Conception d'un programme H/BZR

Un programme H/BZR est conçu sous la forme d'un ensemble de blocs appelés nœuds. Un nœud possède des flots d'entrée et des flots de sortie. Il contient des équations qui définissent les sorties en fonction des entrées, des variables locales et éventuellement des variables d'états intermédiaires. Ces équations peuvent être encapsulées dans les états d'un ou de plusieurs automates et peuvent également instancier d'autres nœuds. Chaque nœud peut être équipé d'un *contrat* pour spécifier un ensemble d'objectifs à réaliser, en utilisant la synthèse de contrôleurs discrets.

Automate Un automate possède un ensemble d'états, l'un d'eux étant l'état initial, et des transitions entre eux. Les états sont associés à des équations qui donnent des valeurs aux flots de sortie du nœud qui contient l'automate. La valeur d'un flot de sortie doit être définie à chaque instant. Une transition est associée à une expression booléenne qui peut être liée à un ou plusieurs flots d'entrée du nœud.

À chaque instant, un état est actif et les expressions booléennes associées à ses transitions sortantes sont évaluées, l'une après l'autre suivant l'ordre de déclaration des transitions. Lorsqu'une expression booléenne évaluée est vraie, la transition associée est déclenchée. Lorsqu'une transition est déclenchée, l'état actif est changé de

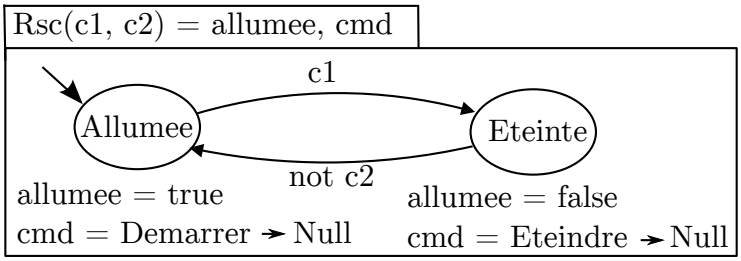

FIGURE 2.6 – Automate modélisant le contrôle d'une ressource de calcul

façon instantanée et les flots de sortie prennent les valeurs données par les équations du nouvel état actif. Si aucune transition n'est déclenchée, l'état actif reste inchangé.

La Figure 2.6 présente un exemple d'automate qui modélise le contrôle d'une ressource de calcul. Cet automate est contenu dans un nœud qui possède deux flots d'entrée (`c1`, `c2`) et deux flots de sortie (`allumee`, `cmd`). L'automate possède deux états (`Allumee`, `Eteinte`) et deux transitions. Chaque état est associé à deux équations qui donnent des valeurs aux flots de sortie. À l'état `Allumee`, le flot de sortie `allumee` est égal à `true`. Cela signifie que la ressource de calcul est allumée. Le flot de sortie `cmd` est égal à `Demarrer` lorsque l'état `Allumee` est nouvellement atteint. Autrement, il est égal à `Null`. La raison est double. D'abord, la valeur d'un flot de sortie doit être définie à chaque instant. Ensuite, cela empêche d'envoyer, de façon continue, `cmd = Demarrer` alors que la ressource de calcul est déjà allumée.

L'état initial de l'automate est `Allumee`. Dans cet état, lorsque la valeur du flot d'entrée `c1` est égale à `true`, l'automate va dans l'état `Eteinte` et les flots de sortie prennent les valeurs données par les équations de cet état. Autrement (la valeur `c1` est égale à `false`), l'automate reste dans l'état `Allumee`. Cela signifie que dans l'état `Allumee`, il existe une transition implicite associée à `not c1` (`c1` = `false`) qui permet de rester dans cet état. De même, lorsque l'automate est dans l'état `Allumee`, si `not c2` est égale à `true` (`c2` = `false`), l'automate va dans l'état `Eteinte`. Sinon (la valeur de `c2` est égale à `true`), l'automate reste dans l'état `Allumee`. Il y a une transition implicite qui est associée à `c2` qui permet de rester dans l'état `Allumee`.

Cet exemple de nœud pourrait être conçu en utilisant un seul flot d'entrée, pour réduire le nombre de variables utilisées. Par exemple, `c1` et `not c2`, associés aux transitions de l'automate pourraient être respectivement remplacés par `c` et `not c`.

Synthèse de contrôleurs discrets avec H/BZR H/BZR permet d'effectuer la Synthèse de Contrôleurs Discrets (SCD) [36], lors de la compilation, à l'aide d'un mécanisme de *contrat*. Un *contrat* est associé à un nœud et est constitué de trois parties : *assume*, *enforce* et *with*. La partie *assume* définit les hypothèses qui sont émises sur le système. La partie *enforce* définit les objectifs à réaliser et la partie *with* définit les variables contrôlables qui seront utilisées pour réaliser les objectifs.

À partir du *contrat*, l'algorithme de SCD explore l'espace d'états du modèle du système et calcule les valeurs possibles des variables contrôlables. Le but est de réaliser les objectifs spécifiés quelles que soient les valeurs des variables non contrôlables.

Par exemple, les flots d'entrée `c1` et `c2` dans l'automate qui modélise le contrôle d'une ressource de calcul (présenté à la Figure 2.6) peuvent être définis comme des variables contrôlables pour allumer ou éteindre la ressource de calcul lorsqu'un événement spécifique survient. L'algorithme de SCD va calculer leurs valeurs possibles.

Après la synthèse du contrôleur, plusieurs solutions peuvent être possibles par rapport aux objectifs à réaliser. Par exemple, une ressource de calcul dans une pièce peut être allumée ou éteinte lorsqu'une présence n'y est pas détectée. Cependant, une seule solution doit être choisie. Pour ce faire, le *backend* du compilateur H/BZR sélectionne une des solutions. Il est possible de guider la sélection avec deux options.

D'abord, le *backend* du compilateur favorise la valeur `true` à la valeur `false` pour une variable contrôlable booléenne. Par exemple, dans l'automate modélisant le contrôle d'une ressource de calcul, présenté à la Figure 2.6, pour favoriser l'état `Eteinte`, la transition qui va de l'état `Allumee` à l'état `Eteinte` est associé à `c1`. Dans ce cas, la transition implicite qui permet de rester dans l'état `Allumee` est associée à `not c1` et est, par conséquent, défavorisée par le *backend* du compilateur.

La deuxième option est que le *backend* du compilateur suit l'ordre de déclaration des variables contrôlables et leur donne la valeur `true`. Si cela ne réalise pas les objectifs considérés, le *backend* du compilateur donne la valeur `false` aux variables suivant l'ordre inverse de leur déclaration. Par conséquent, en déclarant une variable contrôlable `c1` avant une autre, `c2`, si deux transitions `T1` et `T2`, respectivement, associées à `c1` et à `not c2` sont possibles, le *backend* du compilateur choisira `T1`.

La Figure 2.7 présente un exemple de nœud avec un *contrat* pour éteindre les trois ressources de calcul d'une pièce lorsqu'une présence n'y est pas détectée. Ce nœud possède un flot d'entrée (`c`) et six flots de sortie (`cmd_rsc12`, `cmd_rsc13`, `cmd_rsc14`, `allumee_rsc12`, `allumee_rsc13`, `allumee_rsc14`). Les flots de sortie spécifient les commandes à envoyer aux différentes ressources de calcul et leurs états. Ce nœud définit trois instances du nœud qui modélise le contrôle d'une ressource de calcul (cf. Figure 2.6) et compose leurs automates respectifs en utilisant l'opérateur de composition parallèle `;`. Le *contrat* de ce nœud définit aucune hypothèse (*assume* `true`), un objectif et six variables contrôlables. L'objectif à réaliser est : lorsque la valeur du flot d'entrée `c` est égale à `false` (une présence n'est pas détectée), les trois ressources de calcul de la pièce doivent être éteintes. Les variables contrôlables, quant à elles, correspondent aux flots d'entrée des différentes instances du nœud qui modélise le contrôle d'une ressource de calcul et sont utilisées pour réaliser l'objectif.

Pour un système constitué de nombreuses entités, définir un seul *contrat* pour réaliser les objectifs considérés est limitant. Cela génère un seul contrôleur pour l'ensemble du système. De plus, la synthèse du contrôleur peut prendre beaucoup de temps ou ne pas réussir du fait de limitations de CPU et/ou de RAM. En effet, explorer tout l'espace d'états d'un système constitué d'un nombre élevé d'entités requiert beaucoup de ressources en termes de CPU et de RAM. Pour ces différentes raisons, H/BZR permet d'effectuer de la synthèse de contrôleurs discrets modulaire.

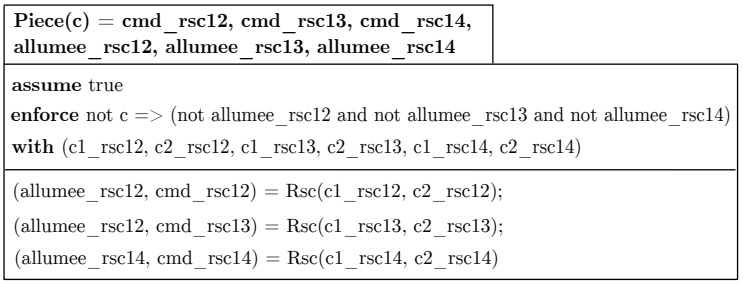

FIGURE 2.7 – Exemple de nœud avec *contrat*

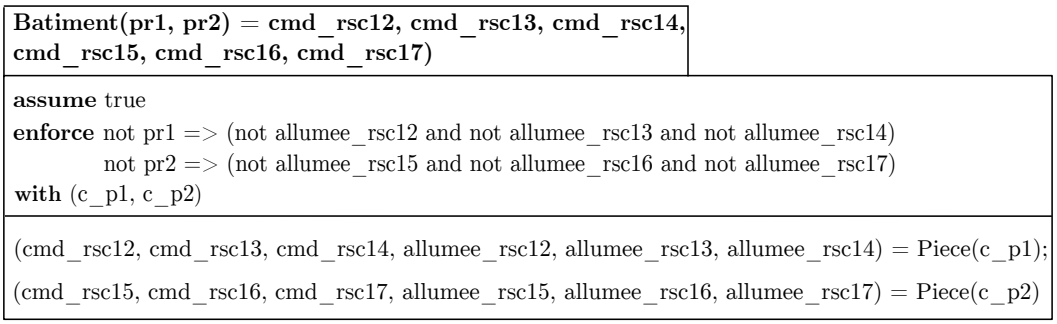

FIGURE 2.8 – Exemple de nœud modulaire

Synthèse de contrôleurs discrets modulaire Le principe consiste à diviser le système considéré en plusieurs sous-systèmes. Chaque sous-système est constitué d'un ensemble d'entités et réalise un certain nombre d'objectifs. Ensuite, à définir pour chaque sous-système, un nœud avec un *contrat* pour réaliser les objectifs du sous-système. Le nœud défini pour un sous-système instancie les nœuds qui correspondent aux différentes entités du sous-système et compose leurs automates. Dans ce cas, la synthèse de contrôleurs discrets est effectuée sur chaque sous-système et non sur l'ensemble du système. Cela diminue le temps d'exécution de l'algorithme de synthèse de contrôleurs discrets et réduit sa consommation de CPU et de RAM.

La Figure 2.8 montre un exemple d'utilisation de la synthèse de contrôleurs discrets modulaire dans H/BZR. Le nœud `Piece` (cf. Figure 2.7), est d'abord instancié deux fois, dans un nœud `Batiment`, pour modéliser un bâtiment de deux pièces contenant chacune trois ressources de calcul. Le nœud `Batiment` prend en entrée deux flots `pr1` et `pr2` modélisant la valeur mesurée par un capteur de présence dans chaque pièce. Ensuite, un *contrat* global est ajouté à ce nœud. L'objectif est d'assurer que, pour chaque pièce, toutes les ressources calcul sont éteintes lorsqu'une présence n'y est pas détectée. Cet objectif est réalisé en utilisant deux variables contrôlables (`c_p1`, `c_p2`) qui sont les flots d'entrée des deux instances du nœud `Piece`. La synthèse de contrôleurs discrets est effectuée sur chacun des trois nœuds.

2.3.3.2 Exécution d'un programme H/BZR

La compilation d'un programme H/BZR génère du code C ou Java. Dans les deux cas, le code généré contient une fonction appelée *step* et une variable appelée *mémoire* contenant l'état de l'automate qui modélise le système considéré. Dans le cas de la Synthèse de Contrôleurs Discrets (SCD) modulaire, plusieurs *step* sont générés (un pour chaque nœud avec un *contrat*) et l'un d'eux est le *step* principal. Le *step* (ou le *step* principal dans le cas de la SCD modulaire) prend comme paramètre les valeurs courantes des entrées du système. Ensuite, il calcule les sorties et met à jour l'état de l'automate, modélisant le système, sous la forme d'une boucle réactive.

Une exécution du *step* (ou du *step* principal dans le cas de la SCD modulaire) correspond à une réaction du système. Par conséquent, le *step* doit être correctement exécuté chaque fois qu'une réaction est requise, en respectant l'hypothèse synchrone. Cela peut être effectué périodiquement ou à chaque fois qu'un événement survient.

2.3.4 Conclusion

Cette section a présenté LINC, PUTUTU et H/BZR. LINC est un intergiciel à base de tuples. Il fournit un langage de règles transactionnelles pour permettre la conception d'applications distribuées et garantir leur fiabilité d'exécution. LINC fournit également des mécanismes pour le déploiement des applications conçues.

PUTUTU est un environnement d'abstraction fournit par LINC. Il est basé sur la mémoire associative pour permettre la communication avec les capteurs et les actionneurs tout en masquant l'hétérogénéité de leurs technologies de communication.

Enfin, H/BZR est un langage de programmation qui est utilisé pour la conception de systèmes réactifs. Il permet d'effectuer, lors de la compilation, la synthèse de contrôleurs discrets pour garantir la fiabilité comportementale des systèmes conçus.

Ces différents outils seront utilisés, par la suite, avec le contrôle continu, la vérification formelle et la vérification de l'exécution des actions effectuées à l'aide des capteurs et sources de données. L'objectif est de permettre la conception et le déploiement de systèmes adaptatifs fiables, sous la forme de boucles autonomiques.

Chapitre 3

Support Intergiciel et Conception d'une boucle autonomique fiable

Sommaire

3.1 Présentation générale de SICODAF 50
3.1.1 Systèmes considérés 50
3.1.1.1 Applications considérées 50
3.1.1.2 Plateformes d'exécution considérées 50
3.1.1.3 Conception et déploiement des systèmes considérés . 51
3.1.2 Support intergiciel SICODAF 51
3.2 Conception d'une boucle générique 53
3.2.1 Définition de la classe de systèmes considérés 53
3.2.2 Flot de conception d'une boucle générique 54
3.2.2.1 Conception de la couche d'abstraction 55
3.2.2.2 Conception des règles d'observation et d'exécution . 55
3.2.2.3 Conception du contrôleur 57
3.2.3 Conception semi-automatique d'une boucle générique 62
3.2.3.1 Génération de la couche d'abstraction 62
3.2.3.2 Génération du contrôleur 62
3.3 Reconfiguration d'une boucle autonomique 63
3.3.1 Principe de la reconfiguration du contrôleur d'une boucle . . 63
3.3.2 Mise en œuvre de la reconfiguration d'une boucle 65
3.4 Intégration d'un système de détection de pannes 65
3.4.1 Exemple d'un système de détection de pannes 65
3.4.2 Mise en œuvre du système de détection de pannes 65
3.4.2.1 Ressources de calcul ou d'entrée/sortie avec une adresse IP 66
3.4.2.2 Ressources d'entrée/sortie sans adresse IP 66
3.4.3 Intégration du système de détection de pannes 67
3.5 Conclusion 67

Ce chapitre présente, dans un premier temps, une vue d'ensemble du support intergiciel, SICODAF, proposé dans cette thèse. Ensuite, il décrit la conception, à l'aide de SICODAF, d'une boucle autonomique générique, pour la mise en œuvre de systèmes adaptatifs fiables. Enfin, ce chapitre montre comment SICODAF permet la reconfiguration d'une boucle et l'intégration d'un système de détection de pannes.

3.1 Présentation générale de SICODAF

Le support intergiciel SICODAF (Support Intergiciel pour la COnception et le Déploiement Adaptatifs Fiables) permet la conception et le déploiement de systèmes adaptatifs, sous la forme d'une boucle autonomique. Cette boucle combine deux formes de fiabilité (une fiabilité comportementale et une fiabilité d'exécution) et peut être reconfigurée, de façon automatique, pour gérer les changements d'objectifs pouvant survenir dans le système. SICODAF permet également l'intégration d'un système de détection de pannes matérielles et la mise en œuvre de boucles multiples pouvant être en parallèle, coordonnées ou hiérarchiques. Cette section décrit les systèmes considérés et le support fourni pour leur conception et leur déploiement.

3.1.1 Systèmes considérés

Les systèmes considérés dans cette thèse sont distribués et adaptatifs. Un tel système est constitué d'une application et d'une plateforme d'exécution. L'application fournit un ensemble de fonctionnalités et est déployée sur la plateforme d'exécution.

3.1.1.1 Applications considérées

Une application est constituée d'un ensemble de tâches de calcul. Une tâche offre une ou plusieurs fonctionnalités et peut utiliser des ressources d'entrée/sortie pouvant être matérielles (p. ex., écran, caméra) ou logicielles (p. ex., base de données). Chaque tâche possède une ou plusieurs versions qui offrent les mêmes fonctionnalités avec des qualités de service différentes (*Elevee*, *Moyenne*, *Faible*). Une tâche peut également avoir plusieurs transitions pour spécifier les changements de versions qui sont valides. Par exemple, considérons une tâche qui possède trois versions : v_1, v_2 et v_3. Pour cette tâche, il peut être valide d'aller de v_1 à v_2 ou de v_2 à v_3 et non de v_1 à v_3 directement, par exemple, pour des raisons de séquencement. Une version de tâche est mise en œuvre sous la forme d'un ensemble d'entités logicielles constituées d'objets et de règles. Les objets et les règles ont des caractéristiques différentes (charges en CPU et RAM). Un objet peut utiliser des ressources d'entrée/sortie. Une règle communique avec un ou plusieurs objets et est exécutée par un objet.

3.1.1.2 Plateformes d'exécution considérées

Une plateforme d'exécution est constituée d'un ensemble de ressources de calcul et de ressources d'entrées/sorties (p. ex., imprimante, scanner) qui sont interconnectées à travers des réseaux. Une ressource de calcul est composée d'un hôte et d'un ensemble de ressources d'entrée/sortie auxquelles elle accède localement ou à travers le réseau. Les hôtes ont des caractéristiques différentes (p. ex., mode de démarrage, capacité en RAM) et des informations de connexion (p. ex., adresse IP, nom d'utilisateur). Les ressources d'entrée/sortie peuvent être matérielles ou logicielles. Une ressource d'entrée/sortie matérielle possède un mode d'utilisation qui spécifie si elle

peut être utilisée par plusieurs tâches et plusieurs objets à la fois. Le mode d'utilisation est : une écriture à la fois ($1E$), plusieurs écritures à la fois (nE) ou lecture (L). Par exemple, une imprimante peut être utilisée, en écriture, par plusieurs tâches à la fois tandis qu'un écran ne peut être utilisé, en écriture, que par une seule tâche à un instant donné. Une ressource d'entrée/sortie matérielle peut avoir une technologie de communication (p. ex., ZigBee [139], EnOcean [42], Plugwise [105]). C'est le cas des capteurs et des actionneurs qui sont installés dans les bâtiments intelligents.

3.1.1.3 Conception et déploiement des systèmes considérés

La mise en œuvre d'un système adaptatif consiste à concevoir l'application associée et la déployer sur une plateforme d'exécution. La conception de l'application consiste à mettre en œuvre les différentes tâches. Le déploiement consiste dans un premier temps à choisir les tâches à activer en fonction des fonctionnalités que le système doit fournir et des ressources d'entrée/sortie disponibles. Ensuite, à choisir pour chacune de ces tâches, la version à activer en fonction de leurs caractéristiques (p. ex., qualité de service fournie, charge en RAM). Enfin, à déployer les objets et les règles des versions choisies sur la plateforme d'exécution. Chaque règle doit être exécutée par un objet et un objet doit être démarré sur une ressource de calcul.

Une fois déployé, le système doit s'adapter aux changements qui surviennent dans son environnement pour rester opérationnel et réaliser ses objectifs. L'adaptation consiste à collecter des données de l'environnement, les analyser pour prendre des décisions d'adaptation et exécuter les actions correspondantes, sous la forme d'une boucle. Les décisions d'adaptation concernent l'application, la plateforme d'exécution ou le déploiement. L'adaptation doit être effectuée de façon autonome et fiable.

3.1.2 Support intergiciel SICODAF

Pour permettre l'adaptation des systèmes, le support intergiciel SICODAF se base sur le calcul autonomique [64]. Le calcul autonomique, comme présenté au chapitre 2, permet la conception de systèmes qui sont capables de s'adapter aux changements de leur environnement de façon autonome. Dans un tel système, comme rappelé à la Figure 3.1, l'adaptation est effectuée par un gestionnaire autonomique sur la base d'une connaissance. Le gestionnaire autonomique observe le système, de façon continue, et collecte des données pour détecter les changements qui surviennent. Ensuite, il analyse les données collectées, prend des décisions d'adaptation et exécute les actions correspondantes, sous la forme d'une boucle autonomique appelée MAPE-K (« *Monitoring*, *Analysis*, *Planning and Execution over a Knowledge* »).

Le support intergiciel SICODAF permet la mise en œuvre de boucles autonomiques fiables pour l'adaptation des systèmes. Une telle boucle est constituée, comme illustrée à la Figure 3.2, d'une couche d'abstraction, d'un mécanisme d'exécution transactionnelle et d'un contrôleur. La couche d'abstraction permet de communiquer avec les différentes entités du système considéré et masque leur hétérogénéité. Le mécanisme d'exécution transactionnelle permet d'exécuter des règles d'observa-

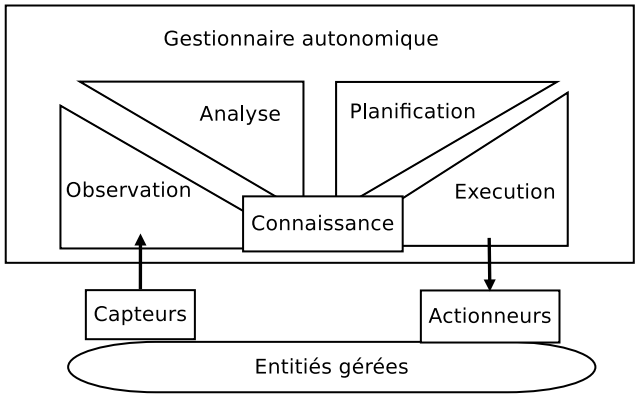

FIGURE 3.1 – Architecture d'un système autonomique

tion et des règles d'exécution. Ces règles collectent des données du système à l'aide de capteurs et exécutent des commandes à l'aide d'actionneurs. Les commandes sont exécutées dans des transactions distribuées pour éviter les inconsistances (le fait de supposer qu'une action est effectuée alors qu'elle ne l'est pas du fait d'une panne matérielle ou d'une erreur de communication). Le contrôleur calcule, en fonction des données qui sont collectées, les commandes qui réalisent les objectifs du système. Une telle boucle peut être une boucle de déploiement ou une boucle applicative.

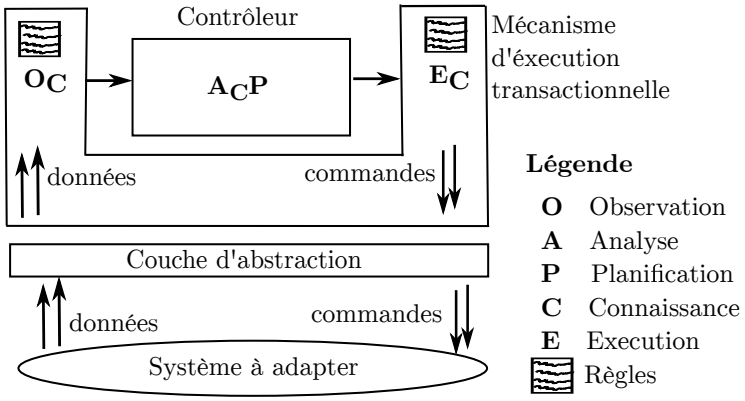

FIGURE 3.2 – Architecture d'une boucle autonomique fiable

La boucle autonomique mise en œuvre pour l'adaptation d'un système peut être reconfigurée de façon automatique. Cela permet de gérer les changements d'objectifs qui surviennent dans le système. Dans ce cas, le contrôleur de la boucle autonomique est remplacé par un autre contrôleur qui réalise les nouveaux objectifs du système.

Le support intergiciel SICODAF permet également l'intégration d'un système de détection des pannes de ressources de calcul et de ressources d'entrée/sortie. L'objectif est de permettre la collecte de données relatives à ces pannes et d'adapter le système considéré en conséquence. Par exemple, le fait de détecter la panne d'une ressource de calcul exécutant des entités logicielles permet de redéployer les entités sur une autre ressource de calcul, afin de continuer à fournir la fonctionnalité cor-

respondante. De plus, le fait de détecter les pannes de ressources de calcul et de ressources d'entrée/sortie permet d'informer la maintenance pour leur réparation.

Enfin, le support intergiciel SICODAF permet la mise en œuvre de boucles multiples pour les systèmes adaptatifs qui sont constitués de nombreuses entités. Dans ce cas, le système considéré est d'abord divisé en plusieurs sous-systèmes. Ensuite, une boucle est mise en œuvre pour chaque sous-système et les boucles sont composées. SICODAF supporte trois modes de composition de boucles : parallèle, coordonné et hiérarchique. Le choix du mode de composition est fait en fonction de la structure du système, des interactions entre les sous-systèmes ainsi que les coûts de conception et d'exécution. La conception de boucles multiples est présentée au chapitre 5.

3.2 Conception d'une boucle générique

Cette section présente comment à partir des systèmes considérés, une boucle générique possédant l'architecture présentée à la Figure 3.2 est conçue. Pour ce faire, cette section définit d'abord la classe de systèmes considérés. Ensuite, elle présente le flot de conception de la boucle générique et la possibilité de la générer.

3.2.1 Définition de la classe de systèmes considérés

Les systèmes considérés sont définis à l'aide d'un méta-modèle d'une application et d'un méta-modèle d'une plateforme d'exécution. Ces méta-modèles décrivent les différentes entités qui constituent les systèmes considérés et identifient leur relations.

La Figure 3.3 présente le méta-modèle d'une application. Une application est composée de tâches qui offrent des fonctionnalités. Une fonctionnalité peut être obligatoire ou optionnelle et possède un niveau de priorité qui est un nombre entier. Le niveau de priorité permet de définir une préférence entre des fonctionnalités qui sont optionnelles. Une tâche peut utiliser des ressources d'entrée/sortie matérielles ou logicielles et possède des versions et des transitions entre les versions. Une version de tâche fournit une qualité de service (p. ex., *Elevee*, *Moyenne* ou *Faible*) et est composée de configurations d'objets et de règles qui sont caractérisées par des charges en CPU et en RAM. Ces charges peuvent être négligeables ou exprimées en pourcentage dans le cas d'une charge CPU ou en *Mo* pour une charge RAM. Une règle communique avec un ou plusieurs objets et est exécutée par au moins un objet. Un objet peut utiliser des ressources d'entrée/sortie et est composé d'un ensemble de configurations d'objets. Chaque entité possède un *id* pour son identification.

La Figure 3.4 présente le méta-modèle d'une plateforme d'exécution. Une plateforme d'exécution est composée de ressources d'entrée/sortie et de ressources de calcul. Une ressource d'entrée/sortie peut être matérielle ou logicielle. Une ressource d'entrée/sortie matérielle possède un mode d'utilisation et peut avoir une technologie de communication. Une ressource de calcul est composée d'un hôte et d'un ensemble de ressources d'entrée/sortie. Elle peut avoir accès à des ressources d'entrée/sortie. Un hôte possède plusieurs caractéristiques (p. ex., système d'exploitation, mode de démarrage, capacité en CPU) et des informations qui permettent de s'y connecter.

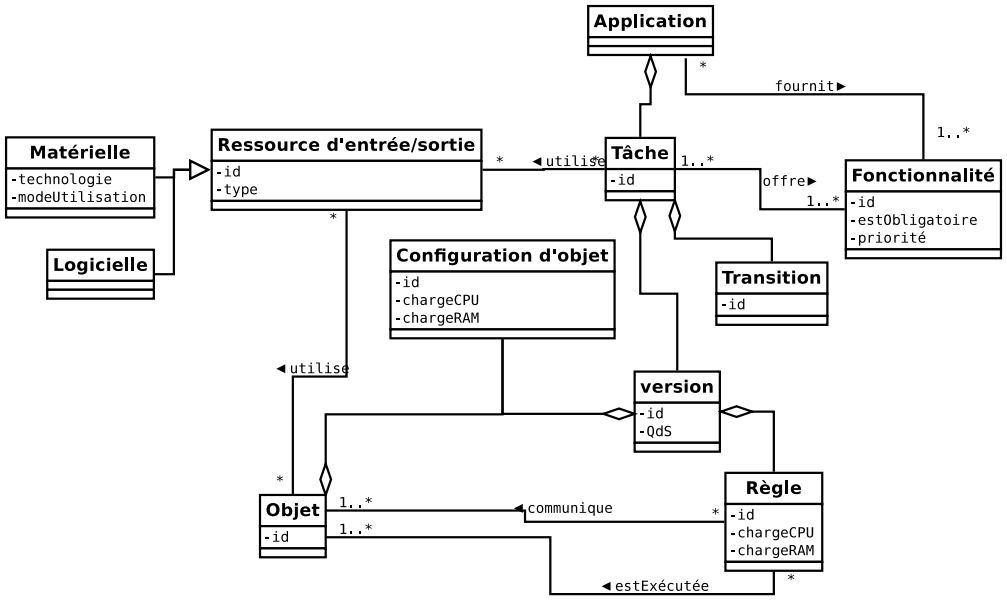

FIGURE 3.3 – Méta-modèle des applications considérées

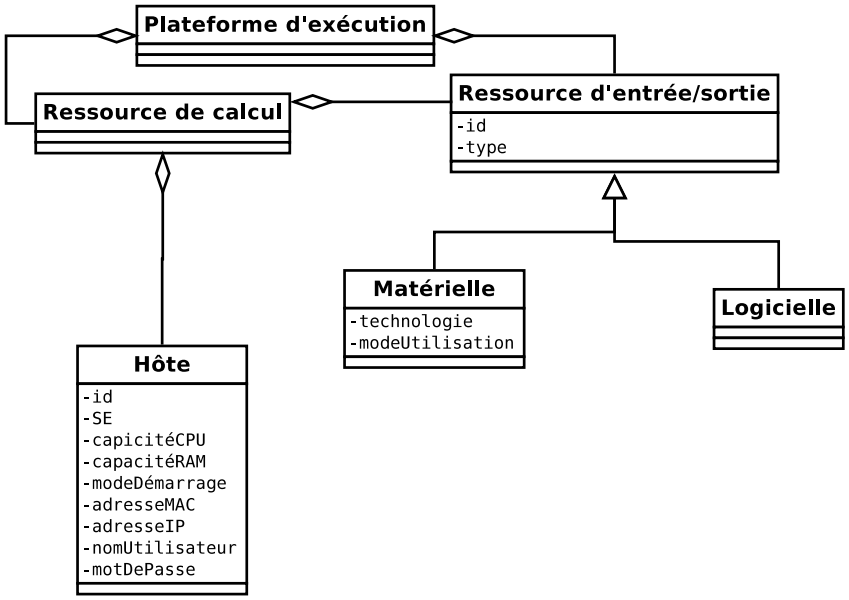

FIGURE 3.4 – Méta-modèle des plateformes d'exécution considérées

3.2.2 Flot de conception d'une boucle générique

Une boucle SICODAF, pour l'adaptation d'un système, est conçue à l'aide :

— **d'un intergiciel à base de tuples avec un langage de règles transactionnelles** donnant accès aux fonctionnalités d'un système d'exploitation ;

— **d'un environnement d'abstraction** qui permet de communiquer avec les différentes entités du système considéré et masquer leur hétérogénéité ;

— **de langages et outils de spécification et de conception de contrôleurs corrects** (p. ex., vérification formelle, synthèse de contrôleurs discrets).

La conception de cette boucle générique consiste en la conception de la couche d'abstraction, des règles d'observation, des règles d'exécution et du contrôleur.

3.2.2.1 Conception de la couche d'abstraction

La couche d'abstraction d'une boucle SICODAF, présentée à la Figure 3.2, est constituée d'un ensemble d'entités logicielles. Certaines de ces entités collectent des données ou exécutent des commandes. Les autres entités sont relatives aux règles d'observation et aux règles d'exécution qui interprètent, respectivement, les données et les commandes. Les entités de la couche d'abstraction sont de deux types :

— les entités qui permettent de communiquer avec les utilisateurs du système à adapter ou les systèmes externes. Un exemple d'utilisateur (resp. de système externe) est l'équipe de maintenance (resp. système de détection de pannes) ;
— les entités qui permettent de communiquer avec le système à adapter.

Entités pour la communication avec les utilisateurs et systèmes externes Ces entités sont conçues à l'aide de l'intergiciel à base de tuples. Pour ce faire, des mémoires associatives dédiées sont créées et exécutées sur des ressources de calcul. Par exemple, une mémoire associative appelée *RequeteMaintenance* peut être créée pour stocker et collecter des requêtes de maintenances pour les ressources de calcul. L'insertion d'un tuple dans cette mémoire associative permet au système de savoir que la ressource de calcul spécifiée doit être éteinte pour une maintenance.

Entités pour la communication avec le système à adapter Ces entités sont spécifiques à chaque type de boucle. Dans le cas d'une boucle de déploiement, elles permettent de communiquer avec l'application à déployer et la plateforme d'exécution. Dans le cas d'une boucle applicative, par exemple pour le bâtiment intelligent, ces entités permettent de communiquer avec les capteurs et les actionneurs qui sont installés dans le bâtiment. Les entités permettant de communiquer avec le système à adapter pour les deux types de boucles sont présentées au chapitre 4.

3.2.2.2 Conception des règles d'observation et d'exécution

La conception des règles d'observation et des règles d'exécution, de la boucle autonomique, de la Figure 3.2, est effectuée à l'aide de l'intergiciel à base de tuples.

Une règle conçue à l'aide de cet intergiciel, comme présentée au Listing 3.1, est constituée de deux parties : *surveillance* et *mise à jour transactionnelle*. Dans la partie *surveillance*, une règle lit une ou plusieurs données sur le système considéré et vérifie, si nécessaire, des conditions sur les données lues. Ensuite, elle exécute une ou plusieurs actions, dans sa partie *mise à jour transactionnelle*. L'aspect transactionnel de la partie mise à jour garantit, dans le cas où la règle effectue plusieurs actions, que toutes les actions sont effectuées ou aucune d'elles n'est exécutée. La lecture d'une

```
Regle [Id]

[Surveillance]

   Donnees
   Conditions

[Mise a jour transactionelle]

   Actions
```

Listing 3.1 – Structure d'une règle

donnée et le test de sa valeur correspondent à une opération de lecture d'un tuple sur une mémoire associative de l'intergiciel à base de tuple. De même, l'exécution d'une action correspond à une opération d'écriture d'un tuple sur une mémoire associative.

Règles d'observation L'objectif de ces règles est d'améliorer l'observation du système considéré. Elles collectent d'abord des données du système à l'aide des capteurs de la boucle autonomique (les entités logicielles qui constituent la couche d'abstraction). Ensuite, ces règles transforment les données collectées en de nouvelles données, qui sont requises par la prise des décisions d'adaptation, et les stockent dans des mémoires associatives dédiées, qui sont au niveau de la couche d'abstraction.

Des exemples de transformations sont : l'agrégation des données qui sont issues de plusieurs capteurs (p. ex., calcul de la moyenne des charges en RAM de deux ressources de calcul qui sont vues comme une seule) ou l'estimation d'une donnée à partir des données des capteurs (p. ex., estimation d'une présence à partir d'une valeur de CO_2). Une règle d'observation est réactive. Elle est déclenchée à chaque fois qu'une nouvelle instance d'une donnée qu'elle collecte est produite dans le système.

Règles d'exécution L'objectif de ces règles est d'améliorer l'exécution des commandes à effectuer sur le système considéré. Ces règles effectuent d'abord une interprétation des commandes, calculées par le contrôleur, en fonction d'une connaissance sur le système et des données qui sont collectées. Ensuite, elles exécutent les commandes sur le système. L'interprétation permet d'identifier, pour chaque commande, les actionneurs de la boucle autonomique qui doivent la recevoir. Une telle règle envoie, selon l'interprétation, une commande à un ou plusieurs actionneurs.

Les règles d'exécution sont basées sur les mémoires associatives de l'intergiciel à base de tuples. La conception d'une règle d'exécution consiste en la redéfinition de l'opération d'insertion de tuples dans une mémoire associative. La redéfinition de cette opération est effectuée pour l'interprétation et l'exécution des commandes. Lorsqu'elle est redéfinie, l'opération lit d'abord des informations (connaissance sur le système, données collectées) relatives à la commande qu'elle reçoit en paramètre. Ensuite, l'opération envoie la commande aux actionneurs qui doivent la recevoir. Par exemple, une règle d'exécution peut être définie pour envoyer une commande à une ressource de calcul qui est constituée de deux ressources de calcul différentes. De

même, une règle d'exécution peut être définie pour identifier les lampes à allumer dans les pièces d'un bâtiment, dans l'objectif de fournir une lumière blanche ou une lumière jaune, en fonction de la préférence des personnes qui occupent les pièces.

3.2.2.3 Conception du contrôleur

Le contrôleur, de la boucle autonomique présentée à la Figure 3.2, calcule, sur la base d'une connaissance sur le système et des données collectées, les commandes à exécuter pour réaliser les objectifs du système. Le contrôleur, comme illustré à la Figure 3.5, possède des entrées, des sorties et il utilise un modèle qui représente les états des entités du système. Le contrôleur peut être conçu de façon manuelle sous la forme d'un ensemble de règles. Le contrôleur peut être également conçu à l'aide de la théorie du contrôle continu [52] ou la théorie du contrôle discret [25] en utilisant un langage de spécification de contrôleurs. Lorsque la théorie du contrôle discret est utilisée, le contrôleur peut être conçu de façon manuelle, puis validé par la vérification formelle, ou de façon déclarative par la synthèse de contrôleurs discrets.

Une fois conçu, le contrôleur de boucle autonomique, est exécuté à l'aide du mécanisme d'exécution transactionnel de l'intergiciel à base de tuples. Le mécanisme d'exécution transactionnel permet d'exécuter les commandes calculées par le contrôleur et la mise à jour de la connaissance du système, dans une transaction, comme une seule opération. Lorsqu'une commande ne peut pas être exécutée du fait d'une panne ou d'une erreur de communication, la transaction échoue et la connaissance sur le système n'est pas mise à jour. Cela permet d'éviter les inconsistances consistant à mettre à jour la connaissance sur le système alors que les commandes n'ont pas été exécutées. Les différents types de contrôleurs sont présentés par la suite.

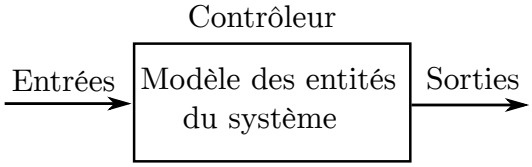

FIGURE 3.5 – Structure d'un contrôleur

Contrôleur basé sur des règles Un contrôleur basé sur des règles est conçu, à l'aide de l'intergiciel à base de tuples. Il est utilisé pour des objectifs qui peuvent être réalisés sous la forme *si alors sinon* et qui impliquent un nombre limité d'entités.

Conception du contrôleur La conception du contrôleur consiste en la définition d'un ensemble de règles qui réalisent les objectifs considérés. Un tel contrôleur possède comme connaissance sur le système, les états logiques des différentes entités du système (c.-à-d. une représentation des états réels des entités). Ces états logiques sont stockés dans des mémoires associatives dédiées de l'intergiciel à base de tuples.

Chaque règle, par exemple celle présentée au Listing 3.2, lit des données sur le système, vérifie des conditions relatives à ces données et effectue des actions. Les

conditions vérifiées par une règle peuvent être relatives à l'occurrence d'événements et aux valeurs des états logiques des entités du système. Une règle est déclenchée lorsque les conditions qu'elle vérifie sont vraies. Les actions effectuées par une règle sont : la vérification des conditions qui l'ont déclenchée, pour voir si elles sont toujours valides, l'exécution de commandes sur le système et la mise à jour des états logiques des entités concernées. La vérification de la validité des conditions, effectuée avant l'exécution des commandes, est motivée par le fait que les systèmes considérés sont dynamiques et l'exécution d'une règle n'est pas instantanée. Il y a un délai entre l'instant où une règle est déclenchée et l'instant où ses actions, relatives à l'exécution des commandes et à la mise à jour des états logiques des entités, sont effectuées. Entre ces deux instants, les conditions qui ont déclenché la règle peuvent ne plus être valides. Ainsi, une vérification des conditions d'entrée est rajoutée dans la transaction qui exécute les commandes et met à jour les états logiques des entités.

Les règles d'un tel contrôleur peuvent être conflictuelles ou violer des objectifs. Par conséquent, la conception du contrôleur requiert la détection et la résolution de conflits et de violations d'objectifs. La résolution est effectuée en vérifiant des conditions additionnelles sur les règles qui sont conflictuelles ou qui violent des objectifs. Le but est d'inhiber certaines règles (empêcher leur activation) lorsque des événements spécifiques surviennent parce qu'elles violent des objectifs du système ou sont en conflit avec d'autres règles. Une règle peut être inhibée si ses objectifs peuvent être réalisés à l'aide d'actions alternatives. Le fait d'inhiber des règles peut rendre nécessaire la définition de nouvelles règles. Ces nouvelles règles réalisent les objectifs des règles qui sont inhibées lorsque les événements considérés surviennent.

Par exemple, considérons un système et deux règles (R_1 et R_2) qui réalisent chacune un objectif du système. Ces règles sont, respectivement, présentées au Listing 3.2 et au Listing 3.3. La règle R_1 vérifie deux conditions (`cond1` et `cond11`), sur le système, et exécute une commande (`cmd1`). La règle R_2 vérifie une condition (`cond2`) et exécute une commande (`cmd2`). Supposons que les commandes exécutées par ces règles sont contradictoires. Dans ce cas, les deux règles sont en conflit lorsque les conditions `cond1`, `cond11` et `cond2` sont vraies. Pour résoudre ce conflit, il faut empêcher le fait que les deux règles soient actives en même temps, en inhibant l'une des règles. Si l'objectif réalisé par R_1 ne peut pas être effectué à l'aide d'une commande alternative et que l'objectif réalisé par R_2 peut être effectué par une autre commande `cmd22`, c'est la règle R_2 qui doit être inhibée. Pour ce faire, des conditions additionnelles relatives à `cond1` et `cond11` sont d'abord rajoutées dans R_2 (cf. Listing 3.4). Ensuite, une nouvelle règle R_3, présentée au Listing 3.5, est définie pour réaliser l'objectif de la règle R_2 lorsque les conditions `cond1` et `cond11` et `cond2` sont vraies. La règle R_3 réalise l'objectif de R_2 en exécutant la commande alternative `cmd22`. La règle R_3 est définie parce que la règle R_2 est inhibée, lorsque les trois conditions sont vraies, et son objectif doit être réalisé (`cond2` est vraie).

```
Regle R1

Surveillance
  Donnees
    % liste des donnees
  Conditions
    cond1 = vrai et cond11 = vrai

Mise a jour transactionelle
  Actions
    Conditions
      cond1 = vrai et cond11 = vrai
    Commandes
      cmd1
    Mise a jour etats logiques
      % liste des operations
```

Listing 3.2 – Exemple de règle : R_1

```
Regle R2

Surveillance
  Donnees
    % liste des donnees
  Conditions
    cond2 = vrai

Mise a jour transactionelle
  Actions
    Conditions
      cond2 = vrai
    Commandes
      cmd2
    Mise a jour etats logiques
      % liste des operations
```

Listing 3.3 – Exemple de règle : R_2

```
Regle R2_modifiee

Surveillance
  Donnees
    % liste des donnees
  Conditions
    cond2 = vrai et (cond1 = faux ou cond11 = faux)

Mise a jour transactionelle
  Actions
    Conditions
      cond2 = vrai et (cond1 = faux ou cond11 = faux)
    Commandes
      cmd2
    Mise a jour etats logiques
      % liste des operations
```

Listing 3.4 – Règle R_2 modifiée

```
Regle R3

Surveillance
  Donnees
    % liste des donnees
  Conditions
    cond2 = vrai et cond1 = vrai et cond11 = vrai

Mise a jour transactionelle
  Actions
    Conditions
      cond2 = vrai et cond1 = vrai et cond11 = vrai
    Commandes
      cmd22
    Mise a jour etats logiques
      % liste des operations
```

Listing 3.5 – Règle R_3

Exécution du contrôleur Les règles qui constituent le contrôleur sont exécutées par un moteur de règles. Pour chaque règle, le moteur évalue d'abord la partie *surveillance* en construisant un arbre d'inférence. Ensuite, il exécute la partie *mise à jour transactionnelle*. L'exécution peut aussi être effectuée de façon distribuée à l'aide de plusieurs moteurs de règles. Ces règles peuvent paraître indépendantes mais elles ne le sont pas. Elles lisent au même instant plusieurs données des mêmes capteurs pour synchroniser leurs actions. Le but est d'éviter les conflits et les violations d'objectifs. Par conséquent, l'exécution distribuée de ces règles peut dégrader la réactivité du système. Elles vont accéder aux mêmes données à travers le réseau. Cependant, cette distribution peut être bénéfique par exemple pour effectuer de la tolérance au pannes ou réduire la charge en CPU et RAM de l'exécution des règles.

Contrôleur basé sur la théorie du contrôle continu Un contrôleur basé sur la théorie du contrôle continu est conçu pour des objectifs dont la réalisation requiert un modèle quantitatif à temps continu qui décrit la dynamique du système considéré. Il peut être également conçu pour la réalisation des objectifs qui consistent à atteindre une valeur de consigne malgré la présence de perturbations extérieures. Un tel contrôleur possède comme connaissance sur le système à adapter, un modèle mathématique qui peut être, par exemple, sous la forme d'équations différentielles.

Conception du contrôleur Le contrôleur est conçu en utilisant les techniques du contrôle continu. Il peut être, selon la nature du système considéré, un contrôleur Proportionnel Intégral Dérivé (PID) ou un contrôleur basé sur les techniques de contrôle avancées (p .ex., commande prédictive, commande H_∞). Par exemple, lorsque le système considéré possède une entrée et une sortie (il est de type SISO *« Single Input Single Output »*) le contrôleur peut être conçu sous la forme d'un PID. Dans ce cas, le contrôleur est d'abord conçu en utilisant l'équation 2.1 du chapitre 2 et en réglant les valeurs des différents gains. Lorsque le système possède plusieurs entrées et plusieurs sorties (il est de type MIMO *« Multiple Input Multile Output »*),

le contrôleur peut être conçu par exemple à l'aide de la commande H_∞. Le contrôleur peut aussi être conçu à l'aide de la commande prédictive, afin d'anticiper le comportement futur du système. Par exemple, dans [91, 132], un contrôleur basé sur la commande prédictive a été conçu afin de minimiser la consommation d'énergie d'un réseau de capteurs tout en respectant les contraintes de l'application. Après sa conception à l'aide des techniques du contrôle continu, le contrôleur est mis en œuvre sous la forme d'une fonction (p. ex., en Matlab ou Python) puis exécuté.

Exécution du contrôleur Pour permettre l'exécution du contrôleur, la fonction associée est invoquée dans une règle. Cette règle est déclenchée, selon le mode de réaction du contrôleur, périodiquement ou à chaque à fois qu'un événement survient dans le système. Lorsque le mode de réaction du contrôleur est périodique, la période de déclenchement de la règle est égale à la période d'échantillonnage du contrôleur. Dans ce cas, le déclenchement périodique de la règle est effectué grâce à l'horloge du système d'exploitation de la ressource de calcul sur laquelle elle s'exécute. Lorsqu'elle est déclenchée, la règle collecte d'abord les données requises. Ensuite, elle invoque le contrôleur pour calculer les commandes à exécuter. Enfin, la règle vérifie si les données collectées sont toujours valides et exécute les commandes calculées.

Contrôleur basé sur la théorie du contrôle discret Un contrôleur basé sur la théorie du contrôle discret est conçu pour des objectifs dont la réalisation consiste à prendre des décisions logiques. Un tel contrôleur est basé sur un système de transitions (p. ex., automates [59], réseau de Petri [94]) qui modélise le système considéré.

Conception du contrôleur Le contrôleur peut être conçu de façon manuelle puis validé par la vérification formelle [125]. Il peut également être conçu de façon déclarative et semi-automatique en utilisant la synthèse de contrôleurs discrets [126].

Contrôleur validé par la vérification formelle : pour concevoir un tel contrôleur, le comportement du système contrôlé est d'abord modélisé sous la forme d'un système de transitions. Ensuite, le modèle est vérifié, à l'aide d'un outil de vérification, pour détecter des conflits et des violations d'objectifs. Lorsque des conflits ou violations d'objectifs sont détectés, le modèle est d'abord modifié pour leur résolution. Ensuite, il est vérifié à nouveau. Par exemple, dans [125], le contrôleur est conçu à l'aide du réseau de Petri coloré [61] puis validé pour la fiabilité comportementale du système. La conception d'un tel contrôleur requiert la spécification manuelle des différentes entités du système considéré et aussi comment elles interagissent pour réaliser les objectifs (les actions à effectuer lorsque des événements surviennent).

Contrôleur basé sur la synthèse de contrôleur discrets : pour concevoir un tel contrôleur, chaque entité du système considéré est d'abord modélisée sous la forme d'un système de transitions en spécifiant ses états, ses transitions d'états et ses caractéristiques. Ensuite, les objectifs que le système doit réaliser sont définis. Les

modèles des entités et les objectifs à réaliser sont utilisés en entrée d'un outil de synthèse de contrôleurs discrets. Cela permet de contrôler le modèle (la composition des modèles des entités) afin de réaliser les objectifs du système. La conception d'un tel contrôleur requiert la spécification des entités du système considéré et des objectifs à réaliser. Elle ne requiert pas de spécifier comment ces entités interagissent pour réaliser les objectifs considérés. Cela est effectué par le contrôleur qui est synthétisé.

Exécution du contrôleur Un contrôleur basé sur la théorie du contrôle discret possède une fonction de transitions. La fonction de transitions d'un contrôleur prend en entrée les données qui sont collectées sur le système. Ensuite, elle déclenche une ou plusieurs transitions, du modèle du système, pour calculer les commandes à exécuter afin de réaliser les objectifs et mettre à jour l'état du modèle du système. Pour permettre son exécution, la fonction de transitions est invoquée dans une règle.

3.2.3 Conception semi-automatique d'une boucle générique

Une boucle générique peut être conçue de façon semi-automatique. Dans ce cas, les composants de la boucle sont générés à partir d'un fichier de description du système considéré et de ses objectifs. Ce fichier est fourni par le développeur.

3.2.3.1 Génération de la couche d'abstraction

Deux mémoires associatives, *RequeteMaintenance* et *Etat*, sont générées pour communiquer, respectivement, avec l'équipe de maintenance et un système de détection de pannes. La mémoire associative *RequeteMaintenance* contient des tuples qui sont sous la forme *(id, requete)* où *id* correspond à l'identifiant d'une ressource de calcul ou d'une ressource d'entrée/sortie et *requete* est une variable booléenne. Lorsque la variable *requete* est égale à vraie, le tuple spécifie que la ressource de calcul ou d'entrée/sortie, identifiée par son *id*, ne doit pas être utilisée du fait d'une opération de maintenance, par exemple une mise à jour matérielle. La mémoire associative *Etat* contient des tuples qui sont sous la forme *(id, panne, disponible)* où *panne* et *disponible* sont des variables booléennes. Un tel tuple spécifie qu'une ressource de calcul ou d'entrée/sortie, identifiée par son *id*, est en panne ou disponible.

D'autres entités logicielles sont générées pour communiquer avec le système à adapter. Ces entités dépendent du type de la boucle et sont présentées au chapitre 4.

3.2.3.2 Génération du contrôleur

La contrôleur est généré sur la base d'un modèle comportemental et d'un *contrat*. Le modèle comportemental décrit, sous la forme de systèmes de transitions, les entités du système à adapter. Le *contrat* spécifie les points de contrôlabilité du modèle comportemental, les propriétés qui doivent être valides sur le modèle et les hypothèses émises. Le modèle comportemental et le *contrat* sont fournis en entrée d'un outil de synthèse de contrôleurs discrets qui génère la fonction de transitions du contrôleur. Cette fonction de transitions est exécutée par une règle qui est générée.

```
[memoireAssociative].lecture(id,valeur)
```

Listing 3.6 – Lecture d'une entrée

Génération de la partie *surveillance* de la règle exécutant la fonction de transitions Cette partie lit les valeurs des entrées de la fonction de transitions du contrôleur et est générée comme suit. Une opération de lecture d'un tuple dans une mémoire associative est générée pour chaque entrée de la fonction de transitions (cf. Listing 3.6). Les noms des mémoires associatives dans lesquelles les valeurs des entrées sont lues et les tuples associés sont calculés de façon automatique. Cela est effectué sur la base des noms des variables qui sont utilisées dans les paramètres d'entrée. Par exemple, pour une entrée dont le nom de la variable associée est `rmaint_D1`, le tuple `("D1", rmaint_D1)` est lu dans la mémoire associative *RequeteMaintenance* qui stocke les requêtes envoyées par l'équipe de maintenance.

Génération de la partie *mise à jour transactionnelle* de la règle exécutant la fonction de transitions Cette partie lit d'abord les valeurs des entrées pour vérifier si elles sont toujours valides. Ensuite, elle invoque la fonction de transitions du contrôleur puis exécute les commandes calculées. Elle est générée comme suit :

1. une opération de lecture d'un tuple dans une mémoire associative est générée pour chaque entrée, de la fonction de transitions, pour lire sa valeur et vérifier si elle est toujours valide ;
2. une opération de lecture d'un tuple dans une mémoire associative encapsulant la fonction de transitions est générée, pour l'invoquer et récupérer les commandes calculées (les valeurs des sorties de la fonction de transitions) ;
3. une opération d'insertion d'un tuple dans une mémoire associative est générée pour chaque sortie, de la fonction de transitions, pour exécuter la commande correspondante.

Les opérations de la partie *mise à jour transactionnelle* de la règle sont embarquées dans une transaction. Le but est d'éviter les inconsistances et le fait d'exécuter des commandes lorsque les entrées qui ont déclenché la règle ne sont plus valides. Une fois générées, les entités de la couche d'abstraction et la règle exécutant la fonction de transitions du contrôleur peuvent être exécutées pour l'adaptation du système.

3.3 Reconfiguration d'une boucle autonomique

Cette section présente la reconfiguration du contrôleur d'une boucle autonomique. Elle décrit le principe de la reconfiguration et présente sa mise en œuvre.

3.3.1 Principe de la reconfiguration du contrôleur d'une boucle

La reconfiguration du contrôleur d'une boucle permet de gérer les changements d'objectifs qui surviennent dans le système. Elle consiste d'abord à définir pour

une boucle, plusieurs contrôleurs qui réalisent des objectifs différents. Ensuite, à remplacer, comme illustré à la Figure 3.6, le contrôleur de la boucle par un autre contrôleur lorsque des événements spécifiques surviennent. Le remplacement est effectué en désactivant le contrôleur courant et en activant un autre contrôleur.

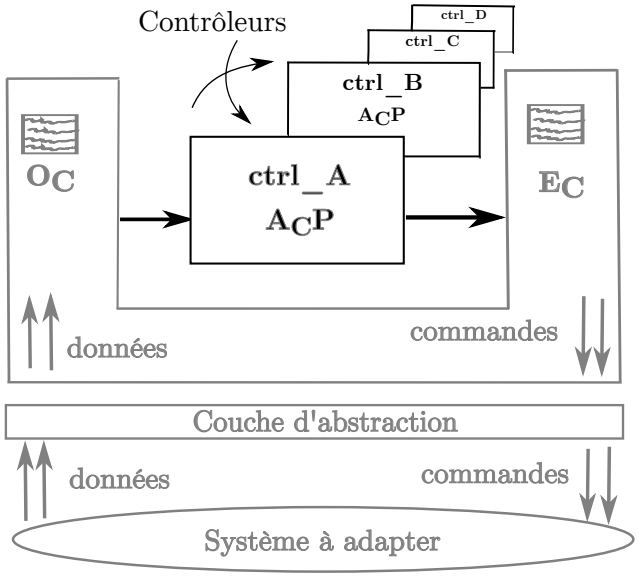

FIGURE 3.6 – Principe de reconfiguration d'une boucle autonomique

Lorsque le contrôleur à activer est basé sur la théorie du contrôle discret, la reconfiguration requiert de garantir que l'état courant du système est un état valide du contrôleur à activer. Un état valide pour un contrôleur (p. ex., `ctrl_A`) peut être invalide pour un autre contrôleur (p. ex., `ctrl_B`), par exemple parce qu'il n'est pas connu du contrôleur `ctrl_B` ou qu'il viole un objectif devant être réalisé par le contrôleur `ctrl_B`. Un état qui est invalide pour un contrôleur n'est pas autorisé à être atteint lorsque ce contrôleur est actif. Par conséquent, le remplacement du contrôleur d'une boucle par un autre n'est possible que si l'état courant du système, qui est valide pour le contrôleur courant, est un état valide du contrôleur à activer.

Il n'est pas trivial de vérifier si un état est valide pour un contrôleur basé sur un système de transitions. En effet, un état est valide pour un tel contrôleur si :

— il appartient à l'espace d'états (l'ensemble des états connus) du contrôleur ;
— il ne viole pas un ou plusieurs objectifs réalisés par le contrôleur ;
— il ne mène pas, par une ou plusieurs transitions, à un état qui viole un objectif.

Cependant, des solutions particulières peuvent être utilisées pour permettre l'activation d'un contrôleur basé sur un système de transitions. Par exemple, tous les contrôleurs d'une boucle autonomique, basés sur un système de transitions, peuvent être conçus de sorte qu'ils aient le même état initial qui est un état de reconfiguration du système. Il est ainsi possible de changer de contrôleur lorsque le système est dans cet état. Par exemple, un bâtiment peut avoir un état de reconfiguration dans lequel il est : non occupé, complètement fermé, non refroidi et non ventilé. Cet état

peut être utilisé comme état initial de tous les contrôleurs de la boucle autonomique, définie pour ce bâtiment, qui sont basés sur un système de transitions (p. ex., un contrôleur pour les périodes de travail et un contrôleur pour les vacances). Cela permet, par exemple, de désactiver le contrôleur des périodes de travail et d'activer le contrôleur des vacances lorsque le bâtiment est dans son état de reconfiguration.

3.3.2 Mise en œuvre de la reconfiguration d'une boucle

La mise en œuvre de la reconfiguration du contrôleur d'une boucle autonomique consiste en la conception, par le développeur, d'une boucle de reconfiguration. Une boucle de reconfiguration possède comme connaissance, sur la boucle à reconfigurer, un modèle qui représente les états des différents contrôleurs (c.-à-d. *actif* ou *inactif*).

Le contrôleur d'une boucle de reconfiguration prend des décisions logiques relatives à des contrôleurs (c.-à-d. *activer* ou *désactiver*) et peut être conçu de façon manuelle sous la forme de règles. Il peut être également conçu à laide de la théorie du contrôle discret. La conception est effectuée comme présentée à la section 3.2.

3.4 Intégration d'un système de détection de pannes

Cette section illustre la possibilité d'intégrer dans une boucle autonomique, conçue à l'aide du support intergiciel SICODAF, un système de détection de pannes. Elle décrit d'abord un exemple de systèmes de détection de pannes. Ensuite, elle présente la mise en œuvre du système décrit et son intégration dans une boucle.

3.4.1 Exemple d'un système de détection de pannes

Le système considéré permet de détecter les pannes de ressources de calcul et de ressources d'entée/sortie. Il permet également de détecter le fait qu'une ressource de calcul ou une ressource d'entrée/sortie qui était en panne est devenue disponible. La détection des pannes et des disponibilités permet d'adapter le système considéré lorsque de tels événements surviennent. Les données relatives à ces pannes et disponibilité sont collectées par la boucle autonomique mise en œuvre pour le système.

La détection des pannes et des disponibilités, effectuée par le système considéré, est basé sur un mécanisme de supervision de type *« keep-alive »*. Un signal est envoyé périodiquement aux ressources de calcul et aux ressources d'entrée/sortie de la plateforme d'exécution considérée. Si aucune réponse n'est reçue après le délai d'expiration, la ressource est supposée être en panne. Pour une ressource qui était en panne, le fait de recevoir une réponse signifie que la ressource est devenue disponible.

3.4.2 Mise en œuvre du système de détection de pannes

Le système de détection de pannes est mis en œuvre à l'aide de l'intergiciel à base de tuples. Il est constitué de mémoires associatives pour stocker et lire des données et de règles pour détecter les pannes et les disponibilités. Ces mémoires associatives et règles sont relatives aux ressources de calcul et aux ressources d'entrée/sortie.

3.4.2.1 Ressources de calcul ou d'entrée/sortie avec une adresse IP

Pour détecter les pannes et les disponibilités des ressources de calcul et ressources d'entrée/sortie qui possèdent une adresse IP, deux mémoires associatives et deux règles sont définies. Ces mémoires associatives sont *Information* et $Etat_{rsc}$. La mémoire associative *Information* contient des informations sur les ressources de calcul et d'entrée/sortie, sous la forme *(id, adresse IP, délai d'expiration)*. La mémoire associative $Etat_{rsc}$ contient des tuples sous la forme *(id, état)*. Un tel tuple associe l'*id* d'une ressource à son état qui peut être égal à *éteinte*, *allumée* ou *indisponible*.

Les deux règles vérifient, respectivement, des pannes et des disponibilités, à l'aide de la commande *Ping*. Ces règles sont déclenchées périodiquement à l'aide de l'horloge du système d'exploitation de la ressource de calcul sur laquelle elles s'exécutent. Lorsqu'elle est déclenchée, la règle de détection de pannes lit d'abord, dans la mémoire associative $Etat_{rsc}$, les *id* des ressources qui sont allumées. Ensuite, la règle lit pour chaque ressource allumée, son adresse IP et son délai d'expiration, dans la mémoire associative *Information* à l'aide de son *id*. Ensuite, la règle effectue un *Ping* sur la ressource. Si aucune réponse n'est reçue après le délai d'expiration, la règle décide que la ressource est tombée en panne et met à jour l'état de la ressource dans la mémoire associative *Etat*, en consommant le tuple *(id, "allumée")* et en insérant le tuple *(id, "indisponible")*. Si une réponse est reçue avant le délai d'expiration (c.-à-d. la ressource est toujours allumée), l'exécution de la règle s'arrête. La règle de détection de disponibilités est également déclenchée périodiquement pour voir si les ressources indisponibles sont devenues disponibles, en effectuant un *Ping* sur chacune de ces ressources. Cette règle change l'état d'une ressource, lorsqu'elle est devenue disponible, de *indisponible* à *allumée* dans la mémoire associative $Etat_{rsc}$.

3.4.2.2 Ressources d'entrée/sortie sans adresse IP

Pour les ressources d'entrée/sortie qui ne possèdent pas une adresse IP et qui peuvent recevoir des commandes (les actionneurs), deux règles qui envoient une commande spécifique, appelée *test*, sont définies. Ces règles sont déclenchées périodiquement et elles utilisent le résultat de l'envoi de la commande *test* pour détecter une panne ou une disponibilité. Une mémoire associative $Etat_{act}$ est également définie. Cette mémoire associative contient des tuples sous la forme *(id, état)* (la valeur de *état* est égale à *disponible* ou *indisponible*) et est mise à jour par les règles de détection de pannes et de disponibilités. Ces règles utilisent les entités logicielles qui sont fournies par l'environnement d'abstraction des technologies de communication, de capteurs et d'actionneurs, pour l'envoi de la commande *test* aux actionneurs.

Lorsqu'elle est déclenchée, la règle de détection de pannes lit d'abord les *id* des actionneurs qui sont disponibles. Ensuite, la règle envoie à chacun des ces actionneurs, la commande *test* et met à jour sont état à *disponible* dans la mémoire associative $Etat_{act}$. L'envoi de la commande à un actionneur et la mise à jour de son état sont effectués dans une transaction. Lorsque la commande ne peut pas être envoyée (l'actionneur est toujours indisponible), la transaction échoue, l'état de l'actionneur n'est pas mis à jour et reste égal à *indisponible*. La règle de détection de

disponibilités, quant à elle, envoie la commande *test* aux actionneurs indisponibles.

3.4.3 Intégration du système de détection de pannes

Un système de détection de pannes peut être intégré dans une boucle autonomique conçue à l'aide du support intergiciel SICODAF. L'intégration est effectuée en rajoutant le modèle de fautes correspondant dans le contrôleur de la boucle et en intégrant les entités logicielles du système de détection de pannes dans la couche d'abstraction. Considérons l'exemple du système de détection de pannes. Son intégration dans une boucle, conçue pour un système, est effectuée comme suit :

Addition du modèle de fautes le contrôleur de la boucle est conçu de sorte qu'il puisse réagir aux disponibilités et indisponibilités des ressources de calcul et des ressources d'entrée/sortie. Pour ce faire, ces états sont rajoutés dans la description de chaque ressource de calcul et d'entrée/sortie, dans le modèle comportemental. Ensuite, la règle qui exécute le contrôleur de la boucle est conçue de sorte qu'elle lise des données dans les mémoires associatives $Etat_{rsc}$ et $Etat_{act}$ du système de détection de pannes, pour détecter le fait que certaines ressources de calcul ou d'entrée/sortie ne sont plus disponibles ou sont devenues disponibles. Cette règle insère également des données dans la mémoire associative $Etat_{rsc}$ pour la mettre à jour lorsqu'elle effectue des commandes sur les ressources de calcul ou d'entrée/sortie. Le but est de notifier au système de détection de pannes, les changements d'états des ressources. Par exemple, lorsque la règle allume (resp. éteint) une ressource de calcul, elle met à jour la mémoire associative $Etat_{rsc}$ pour notifier au système de détection de pannes le fait que la ressource est allumée (resp. éteinte) ;

Ajout des entités du système de détection de pannes les mémoires associatives $Etat_{rsc}$ et $Etat_{act}$ sont rajoutées dans la couche d'abstraction.

Lorsque le système de détection de pannes à intégrer n'est pas basé sur des mémoire associatives et des règles, il est intégré dans une boucle, conçue à l'aide de SICODAF, comme suit. Les entités logicielles de ce système sont d'abord encapsulées dans des mémoires associatives. Ensuite, ces mémoires associatives sont rajoutées à la boucle. L'encapsulation des entités logicielles permet l'intégration d'un système de détection de pannes quel que soit le langage avec lequel il a été mis en œuvre.

3.5 Conclusion

Ce chapitre a présenté le support intergiciel SICODAF proposé pour la conception et le déploiement de systèmes adaptatifs fiables. Il a d'abord décrit les systèmes considérés puis il a présenté une vue d'ensemble du support intergiciel proposé.

Le support intergiciel SICODAF est basé sur les principes du calcul autonomique et permet la conception et le déploiement d'un système adaptatif autonome, sous la forme d'une boucle autonomique. Une telle boucle est conçue à l'aide (i) d'un

intergiciel à base de tuples avec un langage de règles transactionnelles, (ii) d'un environnement d'abstraction (iii) des langages de spécification et outils de conception de contrôleurs (p. ex., vérification formelle, synthèse de contrôleurs discrets).

Une boucle autonomique conçue à l'aide du support intergiciel SICODAF est constituée d'une couche d'abstraction, d'un mécanisme d'exécution transactionnelle et d'un contrôleur. La couche d'abstraction masque l'hétérogénéité des entités du système considéré et permet de communiquer avec elles pour collecter des données et exécuter des commandes. Le mécanisme d'exécution transactionnelle permet d'éviter les inconsistances qui peuvent être causées par des erreurs de communication et des pannes matérielles lors de l'exécution des commandes. Enfin, le contrôleur calcule des commandes correctes et cohérentes qui permettent de réaliser les objectifs du système considéré. Le contrôleur peut être conçu de façon manuelle sous la forme d'un ensemble de règles. Il peut également être conçu à l'aide de la théorie du contrôle continu ou du contrôle discret (conception manuelle puis validation par vérification formelle ou conception déclarative en utilisant la synthèse de contrôleurs discrets).

Le support intergiciel SICODAF permet de reconfigurer le contrôleur d'une boucle afin de gérer les changements d'objectifs qui surviennent dans le système. Ce support intergiciel permet également l'intégration dans une boucle d'un système de détection de pannes matérielles. Cela permet à la boucle de collecter les données relatives à ces pannes et d'adapter le système en conséquence. Ce chapitre a présenté un exemple de système de détection de pannes et son intégration dans une boucle.

Une boucle conçue à l'aide de SICODAF peut être une boucle de déploiement ou une boucle applicative. Les différences entre ces deux types de boucles sont : les données collectées, les commandes calculées et leur exécution. La conception d'une boucle de déploiement et d'une boucle applicative est présentée au chapitre 4.

Enfin, le support intergiciel SICODAF permet de gérer des systèmes adaptatifs qui sont constitués de nombreuses entités ou qui requièrent différents types de contrôleurs. Cela est effectué par la conception de boucles multiples qui sont composées en parallèle, coordonnées ou hiérarchiques, comme présenté au chapitre 5.

Chapitre 4

Conception d'une boucle de déploiement et d'une boucle applicative

Ce chapitre montre comment à partir d'une boucle SIOCDAF générique (cf. chapitre 3), une boucle de déploiement et une boucle applicative peuvent être conçues. Il présente d'abord la conception d'une boucle de déploiement. Ensuite, il décrit, dans le contexte du bâtiment intelligent, la conception d'une boucle applicative.

4.1 Conception d'une boucle de déploiement

Cette section présente la conception d'une boucle de déploiement à l'aide du support intergiciel SICODAF. Elle décrit d'abord les spécificités d'une boucle de déploiement par rapport à la boucle générique présentée au chapitre 3. Ensuite, elle présente le flot de conception d'une telle boucle et la possibilité de la générer. Enfin, cette section présente un exemple pour illustrer le flot de conception d'une boucle.

4.1.1 Spécificité d'une boucle de déploiement

Une boucle de déploiement, conçue pour un système, contrôle l'application, la plateforme d'exécution et le déploiement, pour fournir des fonctionnalités. Une telle boucle réalise des objectifs relatifs par exemple à la continuité du fonctionnement et à l'économie d'énergie de la plateforme d'exécution. Les spécificités de cette boucle sont : les données collectées, les commandes calculées et la couche d'abstraction.

4.1.1.1 Données collectées par une boucle de déploiement

Une boucle de déploiement collecte des données des ressources de calcul et des ressources d'entrée/sortie matérielles de la plateforme d'exécution du système. Des exemples de telles données sont les charges en CPU et RAM d'une ressource de calcul. Cette boucle peut aussi collecter des données qui sont fournies par un utilisateur du système, un opérateur de maintenance ou un système de détection de pannes.

4.1.1.2 Commandes d'une boucle de déploiement

Les commandes d'une boucle de déploiement sont relatives aux objets, aux règles, aux ressources de calcul et aux ressources d'entrée/sortie du système considéré.

commande d'un objet : elle est égale à *démarrer* sur une ressource de calcul, *migrer* d'une ressource de calcul vers une autre ou *arrêter* ;

commande d'une règle : elle est égale à *activer* et faire exécuter par un objet particulier ou *désactiver* ;

commande d'une ressource de calcul : elle est égale à *allumer* ou *éteindre* ;

commande d'une ressource d'entrée/sortie logicielle : elle est égale à *installer* sur une ressource de calcul particulière.

Les commandes calculées peuvent être conflictuelles ou violer des objectifs.

Conflits et violations d'objectifs de déploiement Les conflits de déploiement sont relatifs aux objets, aux règles, aux ressources de calcul et aux ressources d'entrée/sortie du système considéré. Les conflits d'une boucle de déploiement sont :

1. *démarrer (resp. migrer)* un objet sur (resp. vers) une ressource de calcul qui est éteinte ou qui est indisponible du fait d'une panne ou d'une maintenance ;
2. *démarrer (resp. migrer)* un objet sur (resp. vers) une ressource de calcul qui ne possède pas toutes les ressources d'entrée/sortie requises par l'objet ;
3. *activer une règle et la faire exécuter* par un objet qui n'est pas démarré ;
4. *utiliser* une ressource de calcul au-delà de sa capacité en RAM et/ou CPU ;
5. *démarrer (resp. migrer)* des objets sur (resp. vers) une ressource de calcul de sorte qu'ils utilisent tous, en écriture, une même ressource d'entrée/sortie dont le mode d'utilisation est égal à *1E* (c.-à-d. une seule écriture à la fois).

Un conflit a comme conséquence sur le système, la violation d'un ou de plusieurs objectifs. La violation d'un objectif peut aussi être la conséquence d'une décision qui n'est pas conflictuelle mais qui est incorrecte par rapport à cet objectif. Par exemple, considérons un objectif qui consiste à ne pas déployer deux objets utilisés pour de la redondance sur une même ressource de calcul. Cet objectif est violé lorsque ces objets sont démarrés (resp. migrés) sur (resp. vers) une même ressource de calcul.

Contraintes liées à l'exécution des commandes Pour le bon fonctionnement du système, la boucle de déploiement doit respecter l'ordre d'exécution des commandes. L'ordre dans lequel les commandes doivent être exécutées est :

— une ressource de calcul doit être allumée avant d'y installer un logiciel ;
— une ressource d'entré/sortie logicielle doit être installée sur une ressource de calcul, avant d'y démarrer ou migrer un objet qui l'utilise ;
— une ressource de calcul doit être allumée avant d'y démarrer ou migrer un objet ;
— une ressource de calcul ne doit être éteinte qu'après avoir effectué la migration vers une autre ressource de calcul, ou l'arrêt, des objets qu'elle exécute.

4.1.1.3 Couche d'abstraction d'une boucle de déploiement

En plus des mémoires associatives qui permettent de communiquer avec les utilisateurs et les systèmes externes, la couche d'abstraction d'une boucle de déploiement contient d'autres mémoires associatives. Ces dernières permettent de communiquer avec les ressources de calcul, les objets et aussi les règles du système considéré.

Communication avec les ressources de calcul Quatre types de mémoires associatives sont fournis par le support intergiciel SICODAF pour permettre la communication avec les ressources de calcul et masquer leur hétérogénéité. Ces types de mémoires associatives sont : *Configuration*, *Commande*, *Notification* et *Charge*. *Configuration* stocke des informations sur les ressources de calcul sous la forme de tuples *(id, système d'exploitation, mode de démarrage, adresse MAC, adresse IP, nom d'utilisateur, mot de passe)*. *Commande* contient des tuples qui sont sous la forme *(id, commande)*. L'insertion d'un tel tuple envoie d'abord la commande spécifiée à la ressource de calcul dont l'identifiant est égal à la valeur de la variable *id*. Ensuite, elle insère dans *Notification* le résultat de l'envoi de la commande. Pour envoyer la commande, l'opération d'insertion lit d'abord une ou plusieurs informations sur la ressource de calcul, dans *Configuration* à l'aide de son *id*, et utilise la commande appropriée. Par exemple, pour installer un logiciel sur une ressource de calcul dont le système d'exploitation est Ubuntu, l'opération utilise la commande *apt-get install*. *Notification* contient des tuples qui sont sous la forme *(id, commande, résultat)* où résultat est une variable booléenne spécifiant si la commande est effectuée ou non. la mémoire associative *Charge* permet de mesurer la charge d'une ressource de calcul et contient des tuples qui sont sous la forme *(id, charge CPU, charge RAM)*.

Une instance de mémoire associative de chaque type (c.-à-d. *Configuration*, *Commande*, *Notification* et *Charge*) est exécutée sur chaque ressource de calcul du système. La mémoire associative de type *Commande* d'une ressource de calcul permet de l'*éteindre* ou d'y *installer* un logiciel mais ne permet pas de l'allumer. La raison est que lorsque la ressource de calcul est éteinte, ses mémoires associatives ne sont pas accessibles. Par conséquent, la mémoire associative de type *Commande* d'une ressource de calcul qui est éteinte ne peut pas être utilisée pour l'allumer. Pour pouvoir allumer les ressources de calcul, une mémoire associative de type *Commande* exécutée sur une ressource de calcul qui est tout le temps allumée doit être utilisée.

Communication avec les objets Un type de mémoire associative appelé *Objet* est fourni par SICODAF et il contient des tuples qui sont sous la forme *(id objet, id configuration, id ressource de calcul, commande)*. L'insertion d'un tel tuple permet, de démarrer un objet identifié par son *id*, avec la configuration spécifiée, sur une ressource de calcul, de le migrer vers une autre ressource de calcul ou de l'arrêter.

Pour démarrer, migrer ou arrêter un objet, l'opération d'insertion de tuples lit d'abord des informations sur la ressource de calcul spécifiée (p. ex., adresse IP). La lecture de ces informations est effectuée dans une mémoire associative de type *Configuration*. Ensuite, l'opération utilise, en fonction de la commande à exécuter, le

mécanisme de déploiement approprié, de l'intergiciel à base de tuples, pour démarrer, migrer ou arrêter l'objet. Dans le cas du démarrage ou de la migration d'un objet, l'opération vérifie d'abord si le code de l'objet est présent sur la ressource de calcul. Si ce n'est pas le cas, l'opération déploie le code de l'objet sur la ressource de calcul.

Une instance de mémoire associative de type *Objet* est exécutée sur chaque ressource de calcul pour pouvoir y exécuter des commandes relatives aux objets.

Communication avec les règles Une type de mémoire associative appelé *Regle* est fourni et contient des tuples qui sont sous la forme *(id règle, id objet, commande)*. L'insertion d'un tel tuple permet, selon la valeur de la *commande*, d'activer une règle et la faire exécuter par l'objet spécifié ou de la désactiver. Une instance de mémoire associative de type *Regle* est exécutée sur chaque ressource de calcul du système.

4.1.2 Flot de conception d'une boucle de déploiement

La conception d'une boucle de déploiement pour un système peut être effectuée de façon manuelle par le développeur comme présenté au chapitre 3 ou de façon semi-automatique, à l'aide d'un outil du support intergiciel SICODAF. Dans tous les cas, il peut être nécessaire que le développeur écrive des règles d'observation et/ou des règles d'exécution pour étendre la couche d'abstraction de la boucle.

La conception d'une boucle de déploiement peut être effectuée de façon semi-automatique lorsque le contrôleur est basé sur la synthèse de contrôleurs discrets. Dans ce cas, le développeur décrit les entités du système considéré et les objectifs qu'il doit réaliser, dans un fichier de description. Ce fichier de description est ensuite utilisé pour générer les différents composants de la boucle de déploiement.

La Figure 4.1 montre comment à partir d'un fichier de description d'un système les composants d'une boucle de déploiement sont générés. Ces composants sont :

- **les entités de la couche d'abstraction** ces entités correspondent à des mémoires associatives fournies par SICODAF sous la forme de librairies ;
- **la fonction de transition d'un contrôleur** cette fonction est générée à l'aide d'un langage de spécification de contrôleurs, basé sur des systèmes de transitions, qui permet d'effectuer la synthèse de contrôleurs discrets ;
- **une règle exécutant la fonction de transitions du contrôleur**, cette règle collecte les données, invoque la fonction et exécute les commandes.

4.1.2.1 Description du système considéré et de ses objectifs

La description du système et de ses objectifs est effectuée par le développeur, à l'aide d'un langage textuel de SICODAF. Ce langage est conçu à partir du méta-modèle des systèmes considérés (cf. chapitre 3). Il permet de décrire, pour un système, l'application, la plateforme d'exécution et les objectifs qui doivent être réalisés.

Description de l'application Une application est décrite par ses fonctionnalités, ses configurations d'objets, ses objets, ses règles, ses tâches et leurs propriétés.

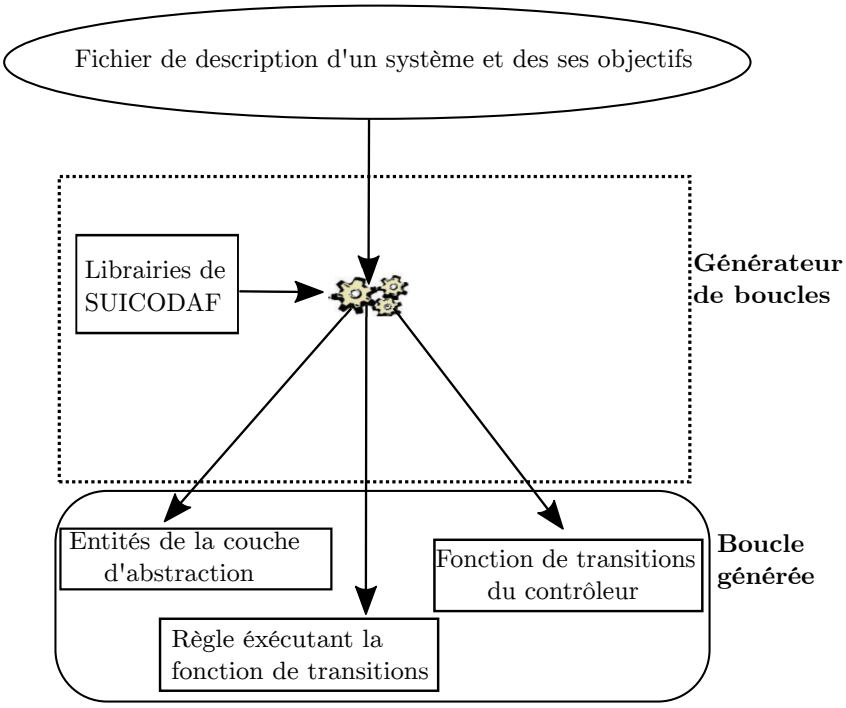

FIGURE 4.1 – Génération d'une boucle autonomique

Description d'une fonctionnalité Une fonctionnalité, comme illustrée au Listing 4.1, est décrite par son identifiant unique et ses propriétés *estObligatoire* et *priorité* qui sont respectivement une variable booléenne et une variable entière.

```
Fonctionnalite [id] : identifiant
  estObligatoire : variable booleene
  priorite : variable entiere
End.
```

Listing 4.1 – Description d'une fonctionnalité

Le Listing 4.2 présente un exemple de fonctionnalité. Cette fonctionnalité a pour *id* `acquisitionDeDonnees`. Elle est obligatoire avec un niveau de priorité égal à 12.

```
Fonctionnalite acquisitionDeDonnees
  estObligatoire : vrai
  priorite : 12
End.
```

Listing 4.2 – Exemple de fonctionnalité

Description d'une configuration d'objet Une configuration d'objets, comme illustrée au Listing 4.3, est décrite par son *id*, sa charge en CPU et sa charge en RAM.

```
Configuration [id] : identifiant
  chargeCPU : variable entiere
  chargeRAM : variable entiere
End.
```

Listing 4.3 – Description d'une configuration d'objets

Le Listing 4.4 présente un exemple de configuration d'objet. Cette configuration a pour *id* `O1.conf1` et possède des charges en CPU et en RAM qui sont négligeables.

```
Configuration O1.conf1
  chargeCPU : 0
  chargeRAM : 0
End.
```

Listing 4.4 – Exemple d'une configuration d'objet

Description d'un objet Un objet, comme illustré au Listing 4.5, est décrit par son *id*, ses ressources d'entrée/sortie requises et ses différentes configurations. Une ressource d'entrée/sortie requise correspond à un type de ressource d'entrée/-sortie. Le nombre de configurations d'un objet doit être supérieur ou égal à un.

```
Objet [id] : identifiant
  ressourcesRequises : liste de types de ressources d'entree/sortie
  configurations : liste de configurations
End.
```

Listing 4.5 – Description d'une configuration d'objet

Le Listing 4.6 présente un exemple d'objet. Cet objet a pour identifiant `O1`. Il requiert le logiciel `Octave` et possède deux configurations `O1.conf1` et `O1.conf2`.

```
Objet O1
  ressourcesRequises : Octave
  configurations : O1.conf1 O2.conf2
End.
```

Listing 4.6 – Exemple d'un objet

Description d'une règle Une règle, comme illustrée au Listing 4.7, est décrite par son *id*, les objets avec qui elle communique, sa charge en CPU et en RAM.

```
Regle [id] : identifiant
  objets : liste d'Objet
  chargeCPU : nombre entier
  chargeRAM : nombre entier
End.
```

Listing 4.7 – Description d'une règle

Le listing 4.8 présente un exemple de règle. Cette règle a pour *id* `R1` et communique avec l'objet `O1`. Cette règle a des charges, en CPU et RAM, négligeables.

```
Regle R1
  objets : O1
  chargeCPU : 0
  chargeRAM : 0
End.
```

Listing 4.8 – Exemple d'une règle

Description d'une tâche Une tâche, comme illustrée au Listing 4.9, est décrite par son *id*, les fonctionnalités qu'elle offre, les ressources d'entrée/sortie qu'elle requiert, ses différentes versions et ses transitions de versions. Le nombre de versions doit être supérieur ou égal à un. Chacune des versions est décrite par son id, ses configurations d'objets et ses règles. Une version doit avoir au moins une configuration d'objet et peut ne pas avoir de règles. Une transition est décrite par son id, une chaîne de caractères formée des *id* de deux versions séparés par le symbole ->.

```
Tache [id] : identifiant
  fonctionnalites : liste de Fonctionnalite
  ressourcesRequises : liste de type de ressources d'entree/sortie
  versions : liste de Version
    Version [id] : chaine de caracteres
     QdS : Elevee| Moyenne| Faible
     configurations : liste de configurations d'objets
     regles : liste de Regle
    End.
  transitions : liste de chaines de caracteres
End.
```

Listing 4.9 – Description d'une tâche

Le Listing 4.10 présente un exemple de tâche. Cet tâche a pour identifiant `T1`. Elle offre deux fonctionnalités `acquisitionDeDonnees` et `analyseDeDonnees` et requiert une ressource d'entrée/sortie (le logiciel `Octave`). Cette tâche a deux versions `T1.V1` et `T1.V2` qui offrent des qualités de services différentes. Chacune des versions est mise en œuvre à l'aide d'une configuration d'objets et d'une règle. `T1` a deux transitions qui permettent d'aller d'une version à une autre (`T1.V1->T1.V2` et `T1.V1->T1.V2`).

```
Tache T1
  fonctionnalites : acquistionDeDonnees analyseDeDonnees
  ressourcesRequises : Octave
  versions :
    Version T1.V1
     QdS : Elevee
     configurations : O1.conf1
     regles : R1
    End.
    Version T1.V2
     QdS : Moyenne
     configurations : O1.conf2
     regles : R2
    End.
  transitions : T1.V1->T1.V2 T1.V2->T1.V1
End.
```

Listing 4.10 – Exemple d'une tâche

Description de la plateforme d'exécution Elle consiste en la description des ressources d'entrée/sortie, des ressources de calcul et de leurs propriétés.

Description d'une ressource d'entrée/sortie logicielle Une ressource d'entrée/sortie logicielle, comme illustrée au Listing 4.11, est décrite par son *id* et son type.

```
RessourceESLogicielle [id] : identifiant
  type : type de RessourceESLogicielle
End.
```

Listing 4.11 – Description d'une ressource d'entrée/sortie logicielle

Le Listing 4.12 présente un exemple de ressource d'entrée/sortie logicielle. Cette ressource d'entrée/sortie logicielle correspond à la version 4.2.1 du logiciel Octave.

```
RessourceESLogicielle Octave4.2.1
  type : Octave
End.
```

Listing 4.12 – Exemple d'une ressource d'entrée/sortie logicielle

Description d'une ressource d'entrée/sortie matérielle Une ressource d'entrée/sortie matérielle, comme illustrée au Listing 4.13, est décrite par son *id*, son type, son mode d'utilisation et sa technologie de communication si elle en a.

```
RessourceESMaterielle [id] : identifiant
  type : type de RessourceESMaterielle
  modeUtilisation : 1E| nE| L
  technologie : nomTechnologie
End.
```

Listing 4.13 – Description d'une ressource d'entrée/sortie matérielle

Le Listing 4.14 présente un exemple de ressource d'entrée/sortie matérielle. Cette ressource d'entrée/sortie matérielle possède un *id* égal à `P1` et est de type imprimante. Son mode d'utilisation est égal à *nE* (c.-à-d. plusieurs écritures à la fois).

```
RessourceESMaterielle P1
  type : imprimante
  modeUtilisation : nE
End.
```

Listing 4.14 – Exemple d'une ressource d'entrée/sortie matérielle

Le Listing 4.15 présente un exemple de ressource d'entrée/sortie matérielle qui est un capteur de température. Son identifiant est `C1`, son mode d'utilisation est égal à *L* (c.-à-d. lecture) et il utilise la technologie de communication `EnOcean` [42].

```
RessourceESMaterielle C1
  type : capteurTemperature
  modeUtilisation : L
  technologie : EnOcean
End.
```

Listing 4.15 – Exemple d'une ressource d'entrée/sortie matérielle

Description d'une ressource de calcul Une ressource de calcul, comme illustrée au Listing 4.16, est décrite par son *id*, son système d'exploitation, sa capacité en RAM, sa capacité en CPU, son mode de démarrage, son adresse MAC, son adresse IP, son nom d'utilisateur, son mot de passe et ses ressources d'entrée/sortie. Chaque ressource d'entrée/sortie est décrite en spécifiant son *id* et le mode d'accès par lequel la ressource de calcul y accède. Le mode d'accès est égal à `local` ou `reseau`.

```
RessourceCalcul [id] : identifiant
  systeme : nomSysteme
  capaciteCPU : nombre entier (en %)
  capaciteRAM : nombre entier (en Mo)
  modeDemarrage : typeDeModeDemarrage
  adresseMAC : chaine de caracteres
  adresseIP : chaine de caracteres
  nomUtilisateur : chaine de caracteres
  motDePasse : chaine de caracteres
  ressourcesES : liste de (RessourceES, modeAcces)
End.
```

Listing 4.16 – Description d'une ressource d'entrée/sortie matérielle

Le Listing 4.17 présente un exemple de ressource de calcul et ses propriétés.

```
RessourceCalcul D1
  systeme : Ubuntu
  capaciteCPU : 200 %
  capaciteRAM : 2000 Mo
  modeDemarrage : wakeOnLAN
  adresseMAC : "B9:42:EB:00:00:00"
  adresseIP : "195.168.1.2"
  nomUtilisateur : "Adja"
  motDePasse : "azerty325"
  ressourcesES : (P1, reseau)
End.
```

Listing 4.17 – Exemple de ressource de calcul

Description des objectifs Elle consiste, comme illustrée au listing 4.18, à décrire les fonctionnalités à fournir et quand est ce qu'elles doivent être fournies. Une fonctionnalité peut être fournie lorsqu'un événement survient ou en continu.

```
Objectif [id]: idententifiant
  Evenements => Fonctionnalite
End.
```

Listing 4.18 – Description d'un objectif

Le Listing 4.19 présente un exemple d'objectif. Cet objectif, `objectif1`, spécifie que la fonctionnalité `acquisitionDeDonnees` doit être fournie de façon continue.

```
Objectif objectif1
  acquisitionDeDonnees
End.
```

Listing 4.19 – Exemple d'un objectif

Le Listing 4.20 présente un exemple d'objectif spécifiant que lorsque l'événement `heureViste` survient, la fonctionnalité `maintienConfidentialite` doit être fournie.

```
Objectif objectif2
  heureVisite => maintienConfidentialite
End.
```

Listing 4.20 – Exemple d'un objectif

4.1.2.2 Génération des entités de la couche d'abstraction

En plus des mémoires associatives, *RequeteMaintenance* et *Etat*, pour la communication avec l'équipe de maintenance et le système de détection de pannes matérielles, des mémoires associatives sont générées pour les ressources de calcul.

Comme illustré à la Figure 4.2, une instance des types de mémoires associatives *Configuration*, *Notification*, *Charge*, *Commande*, *Objet* et *Règle* est générée et exécutée sur chaque ressource de calcul. Les instances des quatre premiers types de mémoires associatives sont relatives à la ressource de calcul. Les instances de type *Objet* et *Regle* permettent, respectivement, de déployer sur la ressource de calcul, des objets et des règles lorsqu'elle est allumée. L'instance de type *Commande* permet d'éteindre la ressource de calcul ou d'y installer un logiciel mais ne permet pas de l'allumer. La raison est que lorsqu'une ressource de calcul est éteinte, les mémoires associatives qu'elle exécute ne sont pas accessibles et ne peuvent pas être utilisées. Pour pouvoir allumer les ressources de calcul, une mémoire associative de type *Commande* appelée *Commande_ Generale* est générée. *Commande_ Generale* doit être accessible à tout instant pour allumer les ressources de calcul du système. Pour cela, elle est exécutée sur une ressource de calcul qui est toujours allumée.

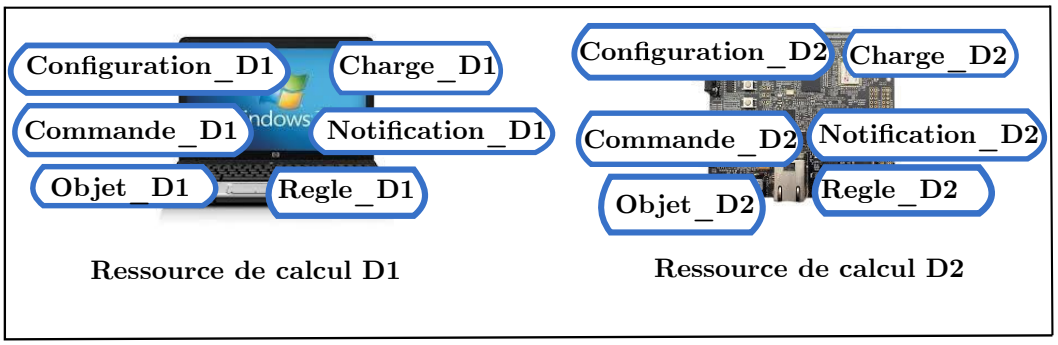

FIGURE 4.2 – Mémoires associatives de la couche d'abstraction

4.1.2.3 Génération de la fonction de transitions du contrôleur

La fonction de transitions du contrôleur est générée, à l'aide de la synthèse de contrôleurs discrets, sur la base d'un modèle comportemental et d'un *contrat*.

4.1.2.3.1 Génération du modèle comportemental Un système de transitions est généré pour chaque tâche, objet, règle, ressource d'entrée/sortie et res-

source de calcul du système considéré. Dans cette section, le système de transitions utilisé, pour illustrer la génération du modèle comportemental, est l'automate.

Automate d'une tâche Une tâche est inactive ou l'une de ses versions est active et offre une qualité de service spécifique. Lorsqu'une version de tâche est active, il peut être possible de changer de version, selon les transitions de versions.

L'automate d'une tâche contient un état *Idle* dans lequel la tâche est inactive. Il contient également un état pour chaque version de la tâche. Chaque état est associé aux configurations d'objets et règles de la version correspondante. Les transitions de cet automate sont les transitions de versions de la tâche. L'automate contient également une transition de l'état *Idle* à chacun des autres états. La raison est qu'une tâche peut être activée avec une de ses différentes versions. L'état *Idle* est préféré aux autres états parce que dans cet état aucune commande ne doit être exécutée. La préférence entre les états qui modélisent les versions de la tâche est basée sur les qualité de services, fournies par les versions associées, dans cet ordre : *Elevee* puis *Moyenne* et ensuite *Faible*. Cette préférence est codée dans l'ordre de déclaration des variables qui sont associées aux différentes transitions de l'automate.

La Figure 4.3 présente l'automate de la tâche `T1` décrite au Listing 4.10. Les entrées de cet automate sont des points de contrôlabilité (leurs valeurs sont définies lors de la synthèse du contrôleur par l'outil de synthèse utilisé). Ces entrées permettent d'activer une version (`c2` et `c3`) ou de désactiver la tâche (`c1`). L'ordre de déclaration des entrées (`c1` puis `c2` puis `c3`) signifie que l'état *Idle* est préféré à l'état `T1.V1` qui est lui préféré à `T1.V2` pour la qualité de service. Les sorties de cet automate correspondent à l'état courant de la tâche (`active`), les configurations d'objets (`configurationsObjets`) et les règles (`regles`) de la version qui est active.

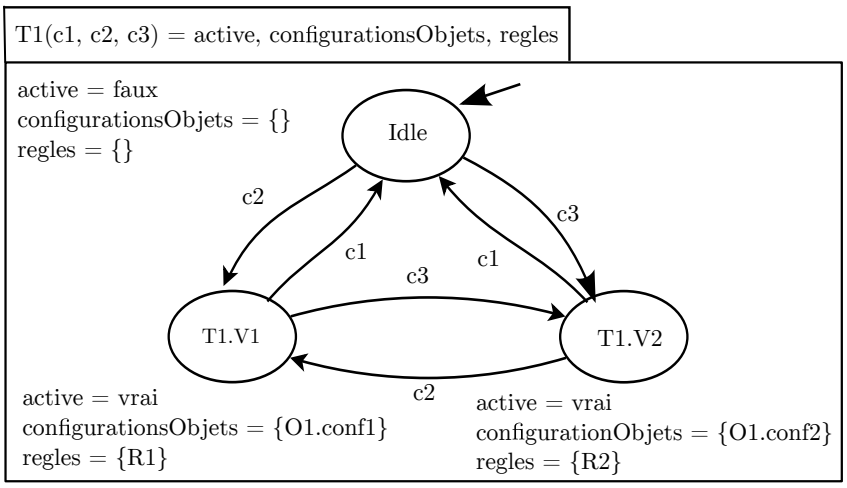

FIGURE 4.3 – Automate de la tache T1

Automate d'un objet Un objet est arrêté ou démarré, en choisissant une de ses configurations, sur une ressource de calcul de la plateforme d'exécution. Lorsqu'il

est démarré, un objet peut être arrêté ou migré vers une autre ressource de calcul.

L'automate d'un objet contient un état *Arrete* et un état pour chaque configuration de l'objet. Un état correspondant à une configuration spécifie que l'objet est démarré avec la configuration associée sur une ressource de calcul particulière.

La Figure 4.4 présente l'automate de l'objet `O1` décrit au Listing 4.6. Les entrées de cet automate sont des points de contrôlabilité. Elles permettent de démarrer (`c2`) l'objet sur une ressource de calcul identifiée par son *id* (`crsc`) ou de l'arrêter (`c1`). Les sorties de cet automate correspondent à l'état de l'objet (`démarre`), sa configuration (`config`), la commande à exécuter (`cmd`) et les identifiants de deux ressources de calcul (`rsc_source`, `rsc_dest`). Ces ressources de calcul correspondent respectivement à la ressource sur laquelle l'objet était démarré et celle sur laquelle il doit être démarré. Par exemple, considérons l'état `Arrete`. Dans cet état, l'objet n'est pas démarré, sa configuration est égale à `configIdle` et la commande est égale à *arreter* la première fois que cet état est atteint sinon elle est égale à `null`. Cela empêche d'envoyer continuellement la commande `arreter` alors que l'objet est déjà arrêté. Dans cet état, la sortie `rsc_dest` est égale à `aucune` (l'objet n'est pas démarré) et la sortie `rsc_source` est égale à sa valeur précédente (`last`) de `rsc_dest` (la ressource de calcul sur laquelle l'objet était exécuté l'instant d'avant). Dans les états où l'objet n'est pas arrêté (`O1.conf1` et `O1.conf2`), la `cmd` est calculée comme suit :

— si `rsc_source` est égale à `aucune` et `rsc_dest` est différente de `aucune` (l'objet était arrêté et doit être démarré), la commande est égale à `demarrer` ;
— si `rsc_source` est égale à `rsc_dest` (l'objet est toujours sur la même ressource de calcul et ne doit pas bouger), la commande est égale à `null` ;
— si `rsc_source` est différente de `rsc_dest` (la ressource de calcul destination a été changée en une autre ressource), la commande est égale à `migrer`.

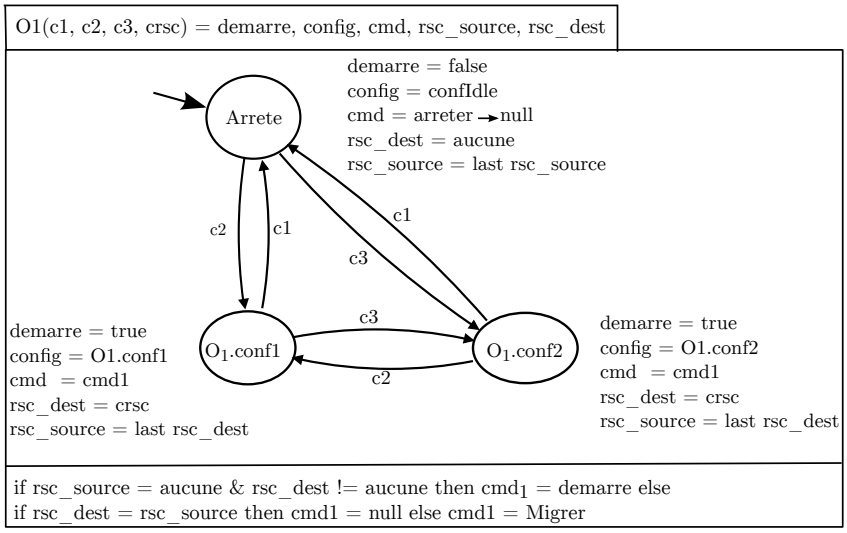

FIGURE 4.4 – Automate de l'objet O1

Automate d'une règle Une règle est désactivée ou activée et exécutée par un objet. L'automate d'une règle contient un état *Desactivee* et un état *Activee*. Cet automate contient deux transitions permettant d'activer et de désactiver la règle.

La Figure 4.5 présente l'automate de la règle `R1` qui est décrite au Listing 4.8. Les entrées de cet automate sont des points de contrôlabilité qui permettent d'activer la règle (`c2`) et de choisir l'objet qui l'exécute (`cobj`) ou de la désactiver (`c1`). Les sorties de cet automate correspondent à l'état de la règle (`active`), la commande à exécuter sur la règle (`cmd`) et l'objet qui a été choisi pour son exécution (`objet`).

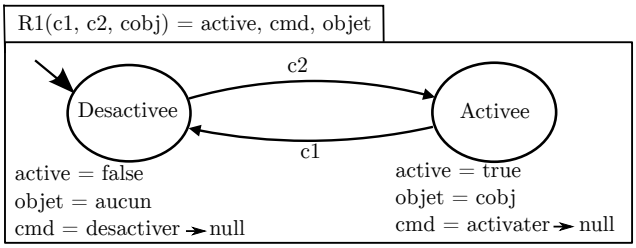

FIGURE 4.5 – Automate de la règle R1

Automate d'une ressource d'entrée/sortie Une ressource d'entrée sortie est indisponible, inutilisée ou utilisée. Elle peut passer de indisponible à inutilisée puis de inutilisée à utilisée. Dans le cas d'une ressource d'entrée/sortie logicielle, elle peut passer de indisponible à utilisée. Dans ce cas, elle est installée puis utilisée.

L'automate d'une ressource d'entrée/sortie contient trois états : *Indisponible*, *Inutilisee*, *Utilisee* et cinq entrées *dispo*, *panne*, *c1*, *c2* et *c3*. Les entrées *dispo* et *panne* permettent de savoir si la ressource d'entrée/sortie est disponible ou est en panne. Dans le cas d'une ressource d'entrée/sortie logicielle, l'entrée *panne* est une constante qui est égale à *false* (un logiciel ne tombe pas en panne). Les entrées *c1* et *c3* sont des points de contrôlabilité qui permettent de changer l'état de la ressource d'entrée/sortie. Dans le cas d'une ressource logicielle, *c2* est un point de contrôlabilité et permet d'installer la ressource logicielle lorsque sa valeur est égale à *false*. Dans le cas d'une ressource d'entrée/sortie matérielle, l'entrée *c2* est une constante qui est égale à *true*. La raison est que la ressource ne peut pas être installée.

La Figure 4.6 présente l'automate de la ressource d'entrée/sortie `Octave4.2.1` décrite au Listing 4.12. Les sorties de cet automate correspondent à l'état de la ressource d'entrée/sortie (`estDispo`, `estUtilisee`) et la commande à exécuter (`cmd`).

Automate d'un hôte Un hôte est éteint, allumé ou indisponible du fait d'une panne ou d'une opération de maintenance. Un hôte peut être dans un état d'attente lorsqu'il est allumé et qu'une requête de maintenance, ne pouvant pas être satisfaite, est reçue. Dans cet état d'attente l'hôte va, dès que possible, à l'état indisponible.

L'automate d'un hôte contient quatre états *Eteint*, *Allume*, *Indisponible*, *Attente*. Il contient six entrées *dispo*, *panne*, *rmaint*, *c1*, *c2* et *c3*. Les trois premières entrées sont fournies par des capteurs. Elles spécifient respectivement si l'hôte est disponible,

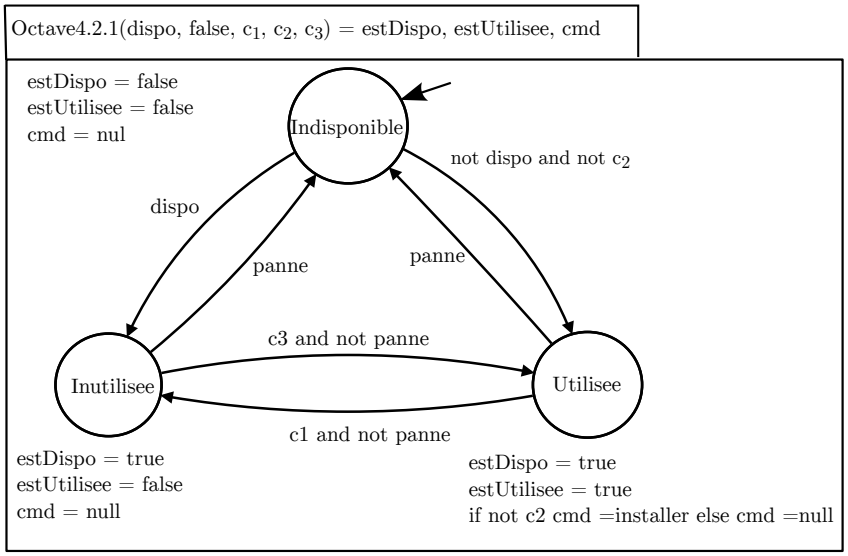

FIGURE 4.6 – Automate de la ressource Octave

s'il est en panne ou une requête de maintenance est envoyée par la maintenance pour ne plus l'utiliser. Les entrées *c1*, *c2*, *c3* sont des points de contrôlabilité qui permettent de démarrer l'hôte, de l'arrêter ou de le mettre dans l'état *Attente*.

La Figure 4.7 présente l'automate de l'hôte associé à la ressource de calcul `D1` décrite au Listing 4.17. Initialement l'hôte est `Etient`. De cet état, il peut être `Indisponible`, suite à une panne ou à la réception d'une requête de maintenance qui est acceptée (`rmaint and c1`), ou `Allume`. Lorsqu'il est `Allume`, l'hôte peut être `Eteint`, `Indisponible` ou aller dans l'état `Attente`. Dans l'état `Attente` l'hôte peut aller à `Indisponible`. Dans l'état `Indisponible`, lorsque l'entrée `dispo` est *true* l'hôte est allumé. S'il n'est pas utilisé pour le déploiement, il est ensuite éteint.

Automate d'une ressource de calcul Une ressource de calcul est composée d'un hôte et de ressources d'entrée/sortie matérielle ou logicielles. Une ressource d'entrée/sortie matérielle est locale à la ressource de calcul (p. ex., un écran) ou est accédée à travers le réseau (p.ex., une imprimante). Une ressource d'entrée/sortie logicielle qui n'est pas disponible sur la ressource de calcul peut être installée.

L'automate d'une ressource de calcul est la composition d'automates suivants :

— **un automate d'hôte** qui modélise l'hôte associé à la ressource de calcul ;
— **un automate de ressource d'entrée/sortie matérielle** pour chaque ressource d'entrée/sortie matérielle qui est locale à la ressource de calcul ;
— **un automate de ressource d'entrée/sortie logicielle** pour chaque ressource d'entrée/sortie logicielle décrite dans le fichier, afin de pouvoir l'installer si elle n'est pas disponible sur la ressource de calcul, en cas de besoin.

La Figure 4.8 présente l'automate de la ressource de calcul `D1`. Cette automate est la composition des automates de l'hôte `H1` (cf. Figure 4.7) et de la ressource d'entrée/sortie logicielle `Octave` (cf. Figure 4.6). L'entrée *dispo* de l'automate de la

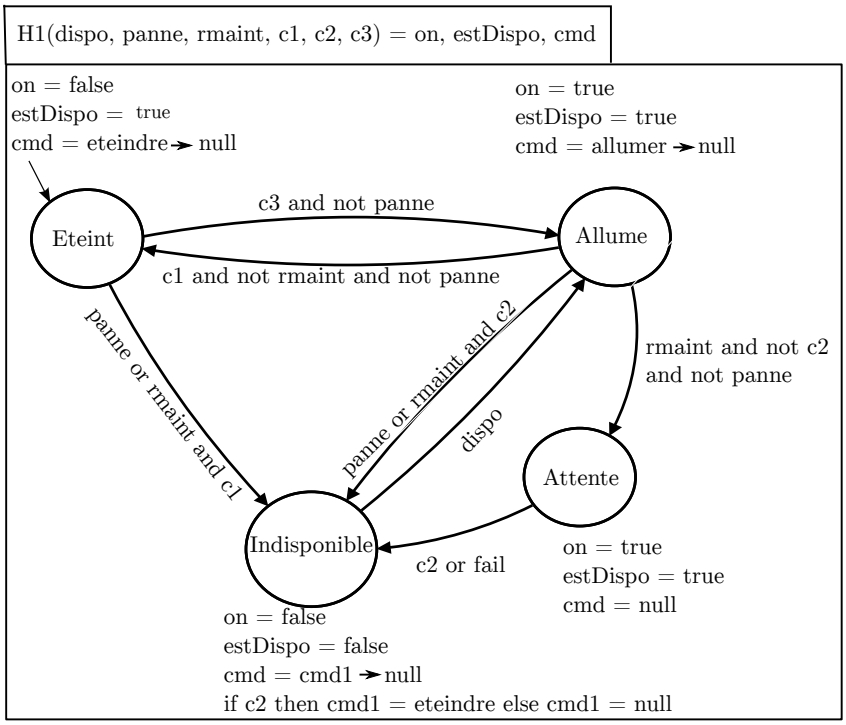

FIGURE 4.7 – Automate de l'hôte de la ressource de calcul Rsc1

ressource `Octave` est égale à `false` car le logiciel Octave ne figure pas dans la liste des ressources d'entrée/sortie de `D1`. `D1` accède à l'imprimante `P1` à travers le réseau. Par conséquent, l'automate de P1 n'est pas pris en compte dans l'automate de `D1`.

```
D1(dispo_H1, panne_H1, rmaint_H1, c1_H1, c2_H1, c3_H1, c1_Octave_D1 , c3_Octave_D1) =
on_D1, estDispo_D1, cmdD1, estDispo_octave_D1, estUtilisee_Octave_D1, cmd_Octave_D1

# automate de l'hôte H1

  (on_D1, estDispo_D1, cmd_D1)  =
   D1(dispo_D1, panne_D1, rmaint_D1, c1_D1, c2_D1, c3_D1);

# automate de la ressource octave

  (estDispo_Octave_D1, estUtilisee_Octave_D1, cmd_Octave_D1) =
   Octave4.21(dispo_octave_D1, false, c1_Octave_D1, c3_Octave_D1)
```

FIGURE 4.8 – Automate de la ressource de calcul D1

Génération du contrat Le *contrat* est constitué de trois parties : hypothèses qui sont émises sur le système considéré, les propriétés qui doivent être valides sur le modèle comportemental et les variables contrôlables du modèle comportemental.

Génération des hypothèses Les hypothèses sont relatives aux ressources de calcul et aux ressources d'entrée/sortie matérielles de la plateforme d'exécution du système considéré. Elles consistent en un modèle de fautes qui permet au système de fonctionner et en dehors duquel il ne fonctionne plus. Ces hypothèses sont :

1. les ressources de calcul ne sont pas toutes indisponibles au même instant ;
2. les ressources d'entrée/sortie matérielles qui possèdent le même type (p. ex., toutes les imprimantes) ne sont pas toutes indisponibles aux mêmes instants. Une ressource d'entrée/sortie qui est locale à une ressource de calcul est indisponible si elle est indisponible ou la ressource de calcul est indisponible ;
3. les ressources d'entrée/sortie utilisées par différentes tâches qui fournissent la même fonctionnalité ne sont pas toutes indisponibles au même instant.

Le développeur peut modifier les hypothèses générées pour le système considéré. Cela lui permet d'utiliser un autre modèle de fautes spécifique au système considéré.

Génération des propriétés Les propriétés permettent de réaliser les objectifs du système. Elles sont relatives aux entités du modèle comportemental et sont :

1. lorsqu'une fonctionnalité doit être fournie, une parmi les tâches qui l'offre doit être active et ses ressources d'entrée/sortie requises doivent être disponibles ;
2. lorsqu'une tâche est active, les objets (resp. les règles) de la version active de la tâche doivent être démarrés sur des ressources de calcul (resp. activées) ;
3. lorsqu'un objet est démarré sur ressource de calcul, elle doit être allumée et doit avoir accès aux ressources d'entrée/sortie qui sont requises par l'objet ;
4. lorsqu'une règle est activée, l'objet choisi pour l'exécuter doit être démarré ;
5. la somme des charges en CPU (resp. des charges en RAM) des objets (les configurations de ces objets) et des règles qui sont déployés sur une ressource de calcul doit être inférieure ou égale à sa capacité en CPU (resp. en RAM) ;
6. une ressource d'entrée/sorite dont le mode d'utilisation est *1E* (une écriture à la fois) doit être utilisée par un seul objet et une seule tâche à la fois.

Génération des variables contrôlables Les variables contrôlables sont les entrées des automates dont les valeurs ne sont pas fournies par des capteurs. Par convention, les noms de ces variables commencent par `c`. Par exemple, les entrées `c1`, `c2` et `c3` dans l'automate d'hôte de la Figure 4.7 sont des variables contrôlables.

4.1.2.4 Génération de la fonction de transitions

Le modèle comportemental et le *contrat* sont utilisés, en entrée d'un outil de synthèse de contrôleurs discrets, pour générer la fonction de transitions du contrôleur de la boucle. Cette fonction de transitions est exécutée par une règle qui est générée.

4.1.2.5 Génération de la règle qui exécute la fonction de transitions

La partie *surveillance* de la règle est générée comme présentée au chapitre 3. La spécificité de cette règle est qu'elle gère l'ordre d'exécution des commandes calculées par le contrôleur. L'ordre d'exécution dépend de la valeur des commandes et n'est pas fixe. Il doit donc être défini de façon dynamique en fonction des commandes. Pour ce faire, la partie *mise à jour transactionnelle* de la règle, dans laquelle les commandes sont exécutées, est générée par une fonction appelée *genererTransaction()* qui est invoquée dans la partie *surveillance* de la règle. Cette fonction est invoquée à chaque fois que de nouvelles entrées sont lues. La raison est que de nouvelles commandes vont être calculées par le contrôleur pour ces entrées. Par conséquent, il faut calculer l'ordre d'exécution de ces commandes. Une fois générée, la transaction est exécutée.

La fonction *genererTransaction()* prend comme paramètre les entrées collectées dans la partie *surveillance* de la règle et génère une transaction qui permet d'exécuter les commandes dans le bon ordre. Une transaction est une règle qui n'a pas de partie *surveillance*. La fonction *genererTransaction()* génère une transaction qui contient :

1. une opération de lecture d'un tuple sur une mémoire associative pour chaque entrée lue dans la partie *surveillance*, pour vérifier si elle est toujours valide ;
2. une opération de lecture d'un tuple sur une mémoire associative encapsulant la fonction de transitions du contrôleur, pour calculer les commandes ;
3. des opérations, d'insertion de tuples dans des mémoires associatives, qui sont ordonnées, pour exécuter les commandes calculées dans le bon ordre.

La fonction *genererTransaction()* définit les noms des mémoires associatives et les tuples dans les opérations générées et aussi l'ordre d'exécution des commandes.

Définition des noms des mémoires associatives et tuples Les noms des mémoires associatives et des tuples sont définis pour les opérations de lecture et les opérations d'écriture de la règle. Pour les opérations de lecture, les noms des mémoires associatives et les tuples, sont définis sur la base des paramètres d'entrées de la fonction *genererTransaction()*. Cela est effectué comme présenté au chapitre 3.

Pour les opérations d'écriture, les noms des mémoires associatives et des tuples sont définis sur la base des commandes calculées. Par exemple, considérons une commande qui est égale à *{"cmd_D1", cmd_D1_val}* où *D1* est une ressource de calcul. Pour cette commande, le tuple inséré est égal à *("D1", cmd_D1_val)* et le nom de la mémoire associative dépend de la valeur de la variable *cmd_D1_val*. Si la valeur de *cmd_D1_val* est égale à *eteindre* alors le nom de la mémoire associative est *Commande_D1* (la mémoire associative de type *Commande* exécutée sur *D1*). Si la valeur de *cmd_D1_val* est égale à *allumer* alors le nom de la mémoire associative est *Commande_Generale*. Cette mémoire associative est celle qui est accessible à chaque instant et utilisée pour allumer toutes les ressources de calcul. *Commande_Generale* est utilisée parce que *Commande_D1* n'est pas accessible. En effet, *D1* est éteinte.

Définition de l'ordre d'exécution des commandes L'ordre d'exécution est généré sur la base des contraintes décrites à la section 4.1.1. Pour ce faire, les commandes relatives aux ressources d'entrée/sortie logicielles et celles relatives aux objets sont d'abord évaluées. L'objectif est d'obtenir pour chacune de ces commandes, sa valeur et les ressources de calcul impliquées. Ensuite, pour chaque ressource de calcul identifiée sa commande est évaluée. Le but est de décider si la commande de la ressource de calcul doit être exécutée avant ou après les commandes de ressources d'entrée/sortie logicielles et d'objets dans lesquelles la ressource de calcul est impliquée. Après l'analyse, l'ordre d'exécution est défini comme suit :

— si une ressource de calcul est impliquée dans l'installation d'une ressource d'entrée/sortie logicielle alors la commande de la ressource d'entrée/sortie logicielle est exécutée avant celle de la ressource de calcul ;
— si une ressource de calcul est impliquée dans le démarrage d'un objet alors la commande de la ressource de calcul est exécutée avant celle de l'objet ;
— si une ressource de calcul est impliquée dans l'arrêt d'un objet, alors la commande de l'objet est exécutée avant celle de la ressource de calcul ;
— si une ressource de calcul est impliquée dans la migration d'un objet et qu'il est la ressource de calcul source (resp. destination), la commande de l'objet est exécutée avant (resp. après) la commande de la ressource de calcul ;
— sur une même ressource de calcul, la commande d'une ressource d'entrée/-sortie logicielle est effectuée avant la commande d'un objet.

4.1.3 Exemple de conception d'une boucle de déploiement

Cette section décrit d'abord un système de traitement de données et ses objectifs et présente la génération d'une boucle de déploiement à partir de cette description.

4.1.3.1 Exemple d'un système de traitement de données

Considérons un système qui est constitué d'une application de traitement de données et d'une plateforme d'exécution. L'application de traitement de données fournit deux fonctionnalités : *acquisition de données* et *analyse de données*. Ces fonctionnalités sont fournies par une tâche $T1$ qui utilise le logiciel Octave. La tâche $T1$ possède deux versions et correspond à la tâche qui est décrite au Listing 4.10. La plateforme d'exécution, associée à l'application de traitement de données, est composée de deux ressources de calcul : $D1$ et $D2$ connectées via un réseau local. Ces ressources de calcul peuvent être démarrées à distance avec le Wake on LAN.

L'application de traitement de données doit être déployée sur la plateforme d'exécution associée. L'objectif est de fournir de façon continue les fonctionnalités *acquisition des données*, *analyse des données*. De plus, après le déploiement, pour rester opérationnel, le système de traitement de données doit s'adapter aux changements qui surviennent (c.-à-d. panne d'une ressource de calcul ou d'une ressource d'entrée/-sortie, requête de maintenance d'une ressource de calcul). Cela est effectué, par la génération, pour ce système à partir de sa description, d'une boucle de déploiement.

4.1.3.2 Génération d'une boucle de déploiement

Huit mémoires associatives (*Commande_ Di*, *Configuration_ Di*, *Notification_ Di* et *Charge_ Di*) sont générées pour les ressource de calcul $D1$ et $D2$. Une mémoire associative *Commande_ Generale* est aussi générée pour pouvoir allumer $D1$ et $D2$. Ensuite, la fonction de transitions du contrôleur est générée à partir d'un modèle comportemental, qui décrit le système de traitement de données, et d'un *contrat*. Cette fonction de transitions est ensuite exécutée par une règle qui est aussi générée.

Modèle comportemental du système de traitement de données Le modèle comportemental décrivant ce système est composé des sept automates :

— un automate de tâches pour $T1$;
— un automates d'objets pour $O1$;
— deux automates de règles pour $R1$ et $R2$;
— deux automates de ressources de calcul pour $D1$ et $D2$.

Les automates de $T1$, $O1$, $R1$ et $D1$ sont ceux qui sont présentés, respectivement, aux Figures 4.3, 4.4, 4.5 et 4.8. Les autres automates du modèle comportemental (ceux de $R2$ et $D2$) sont conçus selon le même principe et ne sont pas présentés.

Contrat du système de traitement de données La Figure 4.9 présente le *contrat* défini pour réaliser l'objectif du système de traitement de données. Cet objectif consiste à fournir de façon continue, les fonctionnalités *acquisition des données*, et *analyse des données*. Le *contrat* est composé de trois parties : *assume*, *enforce* et *with*. La partie *assume* définit l'hypothèse émise sur le système de traitement de données. Cette hypothèse est : l'état où $D1$ et $D2$ sont toutes les deux indisponibles ne peut pas être atteint. La partie *enforce* définit les propriétés qui sont relatives

— **à la tâche** $T1$: elle doit être toujours active car les fonctionnalités *acquisition des données*, et *analyse des données* doivent être fournies de façon continue et elle est la seule tâche qui les offre. Lorsque $T1$ est active, les objets d'une des ses versions doivent être démarrés et les règles associées doivent être actives ;
— **à l'objet** $O1$: lorsqu'il est démarré, la ressource de calcul de destination doit être allumée et doit avoir la ressource d'entrée/sortie logicielle Octave ;
— **aux règles** $R1$ **et** $R2$: pour chacune d'elles, lorsqu'elle est activée, l'objet choisi pour son exécution doit être démarré sur une ressource de calcul.

La partie *with* définit les variables contrôlables. Ces variables correspondent aux points de contrôlabilité des automates qui composent le modèle comportemental.

Le modèle comportemental et le *contrat* sont fournis en entrée d'un outil de synthèse de contrôleurs discrets qui génère la fonction de transitions du contrôleur.

Fonction de transitions du contrôleur La Figure 4.10 présente la fonction de transitions du système de traitement de données. Cette fonction de transitions a six entrées qui correspondent aux événements de panne, de disponibilité et de requête de maintenance relatifs aux ressources de calcul $D1$ et $D2$. Cette fonction de transitions a douze sorties qui correspondent aux commandes à exécuter sur les

```
contract
  assume  not (D1.estDispo = false and D2.estDispo = false)
  enforce
            #propriétés relatives aux tâches
             T1.active and
             T1.active => (T1.configurationObjets.demarre and T1.regles.active)

          # propriétés relatives aux objets
             O1.demarre =>  (O1.rsc_dest.on and O1.rsc_dest.Octave.estDispo)

          # propriétés relatives aux règles
              R1.active => R1.objet.demarre
              R2.active => R2.objet.demarre

  with ( c1_T1, c2_T1, c3_T1, c1_O1, c2_O1, c3_O1,crsc_O1,
         c1_R1, c2_R2, cobj_R1,c1_R1, c2_R2, cobj_R1,
         c1_ocatve_D1, c2_octave_D1, c3_octave_D1,
         c1_ocatve_D2, c2_octave_D2, c3_octave_D1,
         c1_D1, c2_D1, c3_D1, c1_D2, c2_D2, c3_D2)
```

FIGURE 4.9 – *Contrat* du système de traitement de données

entités du système de traitement de données ($D1$, $D2$, $O1$, $R1$, $R2$ et Octave). Cette fonction de transitions est exécutée par la règle qui est présentée au Listing 4.21.

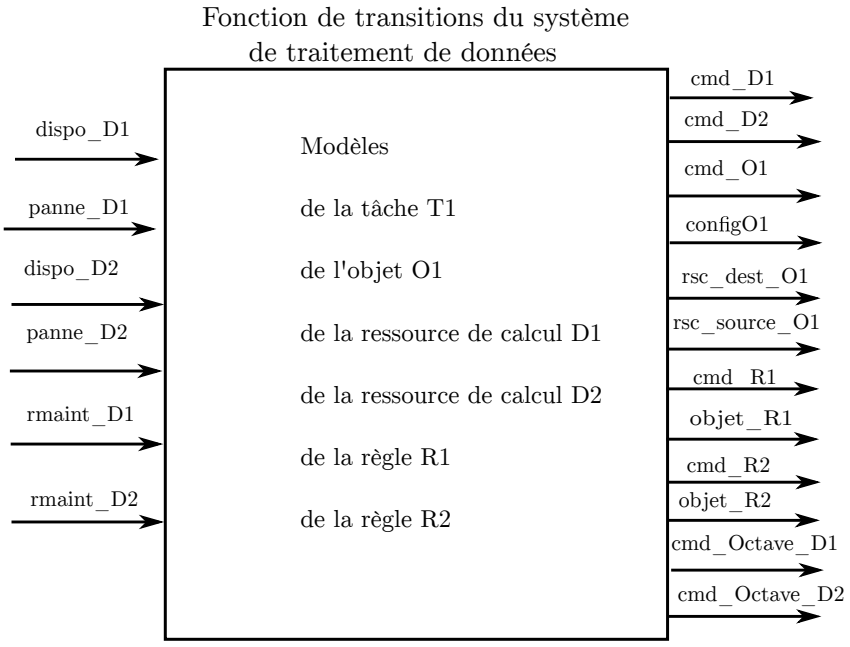

FIGURE 4.10 – *Contrat* du système de traitement de données

```
1  Regle R_traitement_donnees

3  Donnees

5  Etat.lire("D1",panne_D1,dispo_D1)
   Etat.lire("D2",panne_D2,dispo_D2)
7  RequeteMaintenance.lire("D1",rmaint_D1)
   RequeteMaintenance.lire("D2",rmaint_D2)
9  trans = genererTransaction({"panne_D1",panne_D1},{"dispo_D1",dispo_D1},
                              {"panne_D2",panne_D2},{"dispo_D2",dispo_D2},
11                            {"rmaint_D1",rmaint_D1},{"rmaint_D2",rmaint_D2})
   Conditions
13
    % aucune
15
   Actions
17   executer trans
```

Listing 4.21 – Règle exécutant la fonction de transitions traitement de données

4.2 Conception d'une boucle applicative

Cette section présente la conception d'une boucle applicative, dans le contexte du bâtiment intelligent, à l'aide du support intergiciel SICODAF. Elle décrit d'abord les spécificités d'une boucle applicative, conçue pour un bâtiment intelligent, par rapport à la boucle générique présentée au chapitre 3. Ensuite, elle présente le flot de conception d'une boucle applicative et la possibilité de la générer. Enfin, cette section présente un exemple pour illustrer le flot de conception qui a été décrit.

4.2.1 Spécificité d'une boucle applicative de bâtiments intelligents

Une boucle applicative, dans le contexte du bâtiment intelligent, contrôle les capteurs et les actionneurs installés dans le bâtiment considéré. Elle réalise des objectifs définis par le développeur relatifs, par exemple, au confort ou à la confidentialité. Les spécificités d'une boucle applicative, par rapport à la boucle générique présentée au chapitre 3, sont les données collectées, les commandes et la couche d'abstraction.

4.2.1.1 Données collectées par une boucle applicative

Une boucle applicative, dans le contexte du bâtiment intelligent, collecte des données des capteurs qui sont installés dans le bâtiment considéré (p. ex., capteurs de présence, de bruit). Elle peut également collecter des données qui sont fournies par un autre système (p. ex., système de détection de pannes, agenda des occupants). Une boucle applicative peut également collecter des données fournies manuellement par le gestionnaire du bâtiment considéré (p. ex., le début ou la fin des vacances).

4.2.1.2 Commandes d'une boucle applicative

Une boucle applicative, dans le contexte du bâtiment intelligent, calcule des commandes relatives aux actionneurs du bâtiment. Les commandes calculées pour un actionneur dépendent de son type. Des exemples de types d'actionneurs, dans un bâtiment intelligent, sont : des actionneurs de lampes et des actionneurs de volets. La commande calculée pour un actionneur de lampe peut être égale à *allumer* ou *éteindre*. La commande d'un actionneur de volet peut être égale à *ouvrir* ou *fermer*. Les commandes calculées peuvent être conflictuelles ou violer des objectifs à réaliser.

Un conflit survient lorsqu'un même actionneur reçoit, au même instant, des commandes contradictoires. Une violation d'objectif survient lorsque l'exécution d'une commande viole un ou plusieurs objectifs à réaliser. Dans un bâtiment intelligent, les conflits et les violations d'objectifs peuvent être dus à des dépendances qui sont liées aux paramètres de l'environnement. De tels conflits et violations d'objectifs sont implicites et leur détection requiert un modèle de l'environnement. Ce modèle spécifie les effets des actionneurs sur les paramètres de l'environnement. Les conflits et violations d'objectifs sont évités pour la fiabilité comportementale du système.

4.2.1.3 Couche d'abstraction d'une boucle applicative

La couche d'abstraction est conçue en utilisant un environnement qui encapsule différentes technologies de communication de capteurs et d'actionneurs. Des exemples de tels technologies sont : ZigBee [139], EnOcean [42] et Plugwise [105].

Cet environnement d'abstraction fournit des entités logicielles qui permettent de communiquer avec les capteurs et les actionneurs tout en masquant l'hétérogénéité de leurs technologies de communication. Chaque entité logicielle encapsule une technologie de communication et contient une mémoire associative *Sensors* et/ou une mémoire associative *Actuators*. *Sensors* contient des tuples au format *(id, valeur)*. La lecture d'un tel tuple donne la dernière valeur mesurée par le capteur dont l'*id* est spécifié. *Actuators* contient des tuples au format *(id, commande)*. L'insertion d'un tel tuple permet d'envoyer une commande à l'actionneur dont l'*id* est spécifié.

Le Listing 4.22 présente la lecture de la valeur mesurée par le capteur de température *temp_12* qui utilise la technologie de communication *EnOcean*. Le tuple `("temp_12",temp_12_val)` est lu dans la mémoire associative `Sensors` de l'entité logicielle `EnOcean`. Cela retourne dans *temp_12_val* la dernière valeur mesurée.

```
EnOcean.Sensors.lire("temp_12", temp_12_val)
```

Listing 4.22 – Exemple de lectue de la valeur d'un capteur

Le Listing 4.23 présente l'envoi d'une commande, *allumer*, à l'actionneur *chauffage_12* qui utilise la technologie *Plugwise*. Le tuple `("chauffage_12","allumer")` est inséré dans la mémoire associative `Actuators` de l'entité logicielle `Plugwise`.

```
Plugwise.Actuators.ecrire("chauffage12","allumer")
```

Listing 4.23 – Exemple d'envoi d'une commande à un actionneur

La couche d'abstraction de la boucle est constituée d'une entité logicielle pour chaque technologie utilisée par les capteurs et actionneurs du bâtiment considéré.

4.2.2 Flot de conception d'une boucle applicative

Une boucle applicative peut être conçue de façon manuelle, par le développeur, ou de façon semi-automatique à l'aide d'un outil du support intergiciel SICODAF.

La conception semi-automatique d'une boucle applicative est effectuée comme dans le cas d'une boucle de déploiement (cf. Figure 4.1). Les différences sont : la description du système et des objectifs, les entités générées pour la couche d'abstraction et la génération de la règle qui exécute la fonction de transitions du contrôleur.

4.2.2.1 Description du système et de ses objectifs

Le développeur fournit un modèle de l'environnement et définit les objectifs à réaliser. Le développeur identifie dans le modèle de l'environnement les variables contrôlables (points de contrôlabilité) et peut définir des hypothèses sur le système. Enfin, le développeur spécifie pour chaque capteur et chaque actionneur, son type, *id*, sa technologie de communication et son emplacement (intérieur ou extérieur).

Définition du modèle de l'environnement Le développeur considère d'abord un ensemble de paramètres de l'environnement (p. ex., luminosité, bruit, CO_2). Ensuite, il modélise chaque type d'actionneurs sous la forme d'un automate. L'automate d'un type d'actionneurs décrit les différents états du type d'actionneurs, les transitions entre ses états et ses effets sur les paramètres de l'environnement.

Le fait de décrire les types d'actionneurs (p. ex., lampe, volet, fenêtre) et non des actionneurs particuliers (p. ex., lampe_12, volet_15, fenêtre_18), permet la réutilisation du modèle de l'environnement pour un autre système. Par exemple, le modèle défini pour une pièce équipée d'une lampe et d'un volet peut être réutilisé pour une autre pièce qui a deux lampes et deux volets du même type. Un automate d'un type d'actionneurs sera instancié pour chaque actionneur particulier de ce type.

Définition des objectifs à réaliser Pour définir les objectifs, le développeur spécifie les valeurs que les paramètres de l'environnement doivent prendre. La définition des objectifs est effectuée à l'aide de variables et d'opérateurs. Les variables sont relatives aux données collectées et aux paramètres de l'environnement. Les opérateurs, par exemple, $\Rightarrow$ (implication logique), $\wedge$ (et) sont utilisés pour exprimer les relations entre les variables. Des exemples d'objectifs pour une pièce sont :

1. presence $\Rightarrow$ luminosite dans [500,600] lux ;
2. presence $\Rightarrow$ bruit $<$ 80 dB ;
3. presence $\wedge$ temperature $<$ 17 $^\circ$C $\Rightarrow$ heat ;
4. presence $\wedge$ CO_2 $>$ 800 ppm $\Rightarrow$ ventilation.

Le premier objectif spécifie que si une présence est détectée, dans la pièce, la luminosité doit être comprise entre 500 et 600 lux. Le deuxième objectif spécifie que si une présence est détectée, le niveau de bruit doit être inférieur à 80 dB. Le troisième objectif (resp. le quatrième objectif) spécifie que si une présence est détectée dans la pièce et que la température de la pièce est inférieure à 17 °C (resp. le CO_2 est supérieur à 800 ppm), la pièce doit être chauffée (resp. ventilée).

Définition des variables contrôlables et des hypothèses Le développeur identifie les entrées du modèle comportemental qui sont des variables contrôlables et les déclare comme telles. Les entrées du modèle comportemental dont les valeurs sont fournies par les capteurs sont des variables non contrôlable. Les autres sont des variables contrôlables. Le développeur peut également spécifier des hypothèses à émettre sur le système (c-à-d. un modèle de fautes). Un exemple d'hypothèse est : les lampes d'une pièce ne tombent pas toutes en panne au même instant.

4.2.2.2 Génération des entités de la couche d'abstraction

Une instance d'une entité logicielle particulière de l'environnement d'abstraction est générée pour chaque technologie de communication utilisée par les capteurs et/ou actionneurs dont les informations sont décrites par le développeur. Par exemple, si la technologie de communication EnOcean [42] est utilisée par des capteurs et/ou des actionneurs du système, une instance d'une entité logicielle, de l'environnement d'abstraction, qui encapsule la technologie de communication EnOcean est générée.

4.2.2.3 Génération de la règle qui exécute la fonction de transitions

La fonction de transitions est d'abord générée en effectuant la synthèse de contrôleurs discrets sur la description du système et des objectifs. Ensuite une règle *template* est générée pour exécuter la fonction de transitions. Enfin, une instance de la règle *template* qui est spécifique aux capteurs et aux actionneurs décrits est générée.

Génération de la règle template Une règle *template* est indépendante des technologies de communication des capteurs et des actionneurs considérés ainsi que de leurs identifiants. Une telle règle (cf. Listing 4.25) lit des tuples *template* dans des mémoires associatives, *template*, qui sont appelées *memoireAssociative*.

La règle *template* exécutant la fonction de transitions du contrôleur est générée comme présenté à la section 3.2.3 du chapitre 3. La génération d'une règle *template* permet sa réutilisation pour d'autre systèmes du même type, en l'instanciant, pour chacun de ces systèmes, avec les informations des capteurs et actionneurs spécifiques.

Génération de l'instance de règle L'instance de règle est générée en utilisant la règle *template* et la description des informations sur les capteurs et les actionneurs. Cette instance de règle est générée en remplaçant d'abord, dans chaque opération de lecture de la règle template, la variable `memoireAssociative` (resp. `id`) par la

technologie de communication (resp. l'identifiant) du capteur dont la valeur correspond à l'entrée lue. Ensuite, en remplaçant dans chaque opération d'écriture de la règle template, la variable `memoireAssociative` (resp. `id`) par la technologie de communication (resp. l'identifiant) de l'actionneur qui doit recevoir la commande.

4.2.3 Exemple de conception d'une boucle applicative

Cette section décrit d'abord un système de gestion de l'éclairage d'une pièce. Ensuite elle présente la mise en œuvre d'une boucle applicative pour ce système.

4.2.3.1 Exemple de système d'éclairage

Considérons une pièce équipée de deux capteurs (présence, luminosité extérieure) et deux actionneurs (volet, lampe). L'objectif à réaliser est : lorsqu'une présence est détectée dans la pièce, la valeur de la luminosité doit être entre 500 et 600 lux.

4.2.3.2 Description du système d'éclairage et son objectif

Le système d'éclairage est décrit en trois étapes. Les informations (c-à-d. type, *id*, technologie de communication et emplacement) sur les capteurs et actionneurs de la pièce sont d'abord fournies. Ensuite, un modèle de la pièce est défini. Enfin, l'objectif de luminosité à réaliser et les variables contrôlables à utiliser sont définis.

Définition des informations sur les capteurs et actionneurs Le listing 4.24 présente les informations sur les capteurs et actionneurs de la pièce. Ce Listing spécifie, par exemple, que l'*id* du capteur de présence est `pr1` et sa technologie de communication est `TelosB`. Ce capteur est installé à l'intérieur de la pièce.

```
1 # type:id:technologie:emplacement
  presence:pr1:TelosB:interieur
3 luminosity:lu9:RFXCOM:exterieur
  lamp:e_l_1:EnOcean:interieur
5 shutter:volet_43:KNX:interieur
```

Listing 4.24 – Capteurs et actionneurs de l'exemple de la pièce

Définition d'un modèle de la pièce Pour modéliser la pièce, un paramètre d'environnement, *luminosité*, est d'abord défini. Ensuite, pour chaque type d'actionneurs, son effet sur le paramètre *luminosité*, ses états et ses transitions d'états sont décrits, sous la forme d'un automate possédant des entrées et des sorties.

La Figure 4.11 présente la description de l'actionneur de lampe, sous la forme d'un automate. Cet automate a deux entrées (`c1`, `c2`) et deux sorties (`cmd_lampe`, `lum_lampe`). L'automate a deux états (`Eteinte`, `Allumee`) et deux transitions. L'état intial de l'automate est `Eteinte`. Dans cet état, la sortie `cmd_lampe` est égale à `eteindre` si cet état est nouvellement activé et `null` sinon. Cela permet de ne pas envoyer en continu `cmd_lampe` = `eteindre` alors que la lampe est déjà éteinte. Dans l'état `Eteinte`, `lum_lampe` est égale à 0 (lorsqu'elle est éteinte, la lampe ne fournit

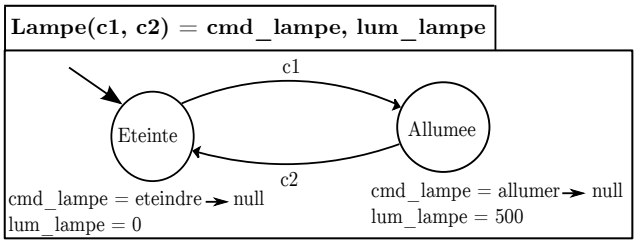

FIGURE 4.11 – Description d'un actionneur de lampe

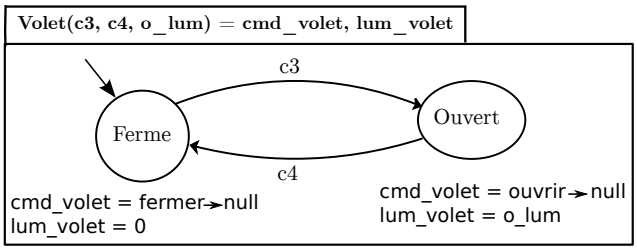

FIGURE 4.12 – Description d'un actionneur de volet

pas de luminosité). Les entrées de l'automate sont des variables booléennes permettant de passer d'un état à un autre. Par exemple, lorsque `c1` est *vraie*, l'automate va dans l'état `Allumee` et les sorties prennent les valeurs qui sont associées à cet état.

La Figure 4.12 présente la description de l'actionneur de volet, sous la forme d'un automate. Cet automate a trois entrées (`c3`, `c4`, `o_lum`) et deux sorties (`cmd_volet`, `lum_volet`). Cet automate a deux états (`Ferme`, `Ouvert`) et deux transitions. L'état initial de l'automate est `Ferme` Dans cet état, le volet ne fournit pas de luminosité. Dans l'état `Ouvert`, il fournit une luminosité égale à la luminosité extérieure (`o_lum`).

Définition des objectifs et des variables contrôlables L'objectif à réaliser dans la pièce est définie comme suit : *i_presence* ⇒ *lum dans [500,600]* où *presence* est une valeur mesurée par un capteur de présence intérieur, *lum* est une variable qui est égale à la somme des luminosités fournies par la lampe et le volet.

Les variables contrôlables correspondent aux variables *c1*, *c2*, *c3* et *c4* qui sont associées aux automates décrivant l'actionneur de volet et l'actionneur de lampe.

4.2.3.3 Génération d'une boucle applicative

La Figure 4.13, présente la fonction de transitions générée. Elle a deux entrées (`i_presence`, `o_lum`) et deux sorties (`cmd_volet`, `cmd_lampe`). Ces entrées correspondent, respectivement, à la valeur mesurée par un capteur de présence intérieur et un capteur de luminosité extérieur. Les sorties sont les commandes à envoyer au volet et à la lampe et sont calculées sur la base d'un modèle d'un volet et d'une lampe. Le Listing 4.25 montre la règle *template* qui exécute la fonction de transitions.

La règle instance générée à partir de la règle *template* (Listing4.25) et de la description des capteurs et actionneurs (Listing 4.24) est présentée au Listing 4.26.

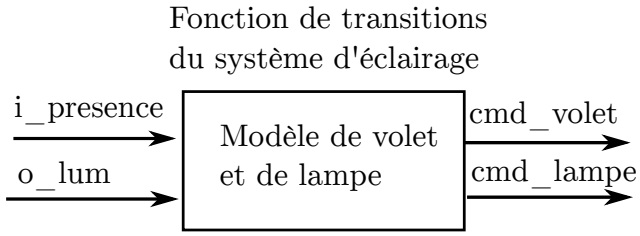

FIGURE 4.13 – Description d'un actionneur de volet

```
1  Regle R_eclairage_template

3  Donnees

5  memoireAssociative.lire(id, i_presence_id_val)
   memoireAssociative.lire(id, i_olum_id_val)
7
   Conditions
9
    % aucune
11
   Actions
13
   memoireAssociative.lire(id, i_presence_id_val)
15 memoireAssociative.lire(id, i_olum_id_val)
   FonctEclairage.lire(i_presence_id_val,i_olum_id_val,cmd_volet, cmd_lampe)
17 memoireAssociative.ecrire(id,cmd_volet)
   memoireAssociative.ecrire(id,cmd_lampe)
```

Listing 4.25 – Règle template exécutant la fonction de transitions éclairage

```
1  Regle R_eclairage_instance

3  Donnees

5  TelosB.Sensors.lire("pr1", i_presence_pr1_val)
   RFXCOM.Sensors.lire("lu9", i_olum_lu9_val)
7
   Conditions
9
    % aucune
11
   Actions
13
   TelosB.Sensors.lire("pr1", i_presence_pr1_val)
15 RFXCOM.Sensors.lire("lu9", i_olum_lu9_val)
   FonctEclairage.lire(i_presence_pr1_val,i_olum_lu9_val,cmd_volet, cmd_lampe)
17 KNX.Actuators.ecrire("volet_43",cmd_volet)
   Enocean.Actuators.ecrire("e_l_1",cmd_lampe)
```

Listing 4.26 – Règle instance exécutant la fonction de transitions éclairage

4.3 Conclusion

Ce chapitre a présenté la conception d'une boucle de déploiement et la conception d'une boucle applicative, dans le contexte du bâtiment intelligent, avec SICODAF.

Une boucle de déploiement, d'un système, contrôle l'application, la plateforme d'exécution et le déploiement afin de fournir une ou plusieurs fonctionnalités. Une boucle applicative quant à elle, dans le contexte du bâtiment intelligent, contrôle les actionneurs dans le but de réaliser des objectifs relatifs, par exemple, au confort.

SICODAF permet la conception manuelle et semi-automatique de ces deux types de boucles. Pour la conception manuelle, un méthodologie est décrite et peut être suive par le développeur. Pour la conception semi-automatique, le développeur fournit un fichier de description du système considéré et des objectifs qu'il doit réaliser.

Dans le cas d'une boucle de déploiement, le développeur utilise un langage textuel. Ce langage a été conçu à partir de la classe de systèmes considérée et permet de décrire, pour un système, l'application, la plateforme d'exécution et les objectifs. Dans le cas d'une boucle applicative, dans le contexte du bâtiment intelligent, le développeur fournit un modèle de l'environnement et définit les objectifs. Le modèle de l'environnement spécifie pour chaque type d'actionneurs (p. ex., lampe, fenêtre) ses états, ses transitions et ses effets sur les paramètres de l'environnement (p. ex., luminosité, bruit), sous la forme d'un système de transitions. Les objectifs sont définis en spécifiant les valeurs que les paramètres de l'environnement doivent prendre.

Pour les deux types de boucles, le fichier de description fourni est utilisé en entrée d'un outil de SICODAF qui génère les différents composants de la boucle. Ces composants sont : (i) la couche d'abstraction à l'aide des librairies fournies par SICODAF, (ii) la fonction de transitions du contrôleur à l'aide d'un outil de synthèse de contrôleurs discrets, (iii) la règle qui exécute la fonction de transitions. Après leur génération, ces composants sont exécutés pour permettre l'adaptation du système.

Chapitre 5

Support pour des boucles multiples

Sommaire

5.1 Motivations et avantages des boucles multiples 97
5.1.1 Motivations 98
5.1.2 Avantages 99
5.2 Modes de composition de boucles 100
5.3 Conception de boucles en parallèle 100
5.3.1 Avantages des boucles en parallèle 100
5.3.2 Limitations des boucles en parallèle 101
5.4 Conception de boucles coordonnées 102
5.4.1 Prérequis de la conception d'un coordinateur 102
5.4.2 Conception d'un coordinateur basé sur des règles 104
5.4.3 Conception d'un coordinateur basé sur le contrôle discret . . 105
5.4.3.1 Coordinateur validé par la vérification formelle . . . 105
5.4.3.2 Coordinateur basé sur la synthèse de contrôleurs discrets 106
5.4.4 Limitations des boucles coordonnées 107
5.5 Conception de boucles hiérarchiques 107
5.5.1 Conception à l'aide du langage de règles 108
5.5.2 Conception à l'aide de la théorie du contrôle discret 108
5.5.2.1 Validation à l'aide de la vérification formelle 109
5.5.2.2 Conception à l'aide de la synthèse de contrôleurs discrets 109
5.6 Conclusion 110

Ce chapitre présente la conception, à l'aide du support intergiciel proposé, de boucles multiples, pouvant être des boucles de déploiement ou des boucles applicatives. Il présente d'abord les motivations et avantages des boucles multiples. Ensuite, il présente trois modes de composition de boucles : parallèle, coordonné et hiérarchique. Enfin, il présente la conception de boucles composées selon chaque mode.

5.1 Motivations et avantages des boucles multiples

La conception de boucles multiples est motivée par la taille des systèmes et aussi par le fait que les objectifs qu'ils doivent réaliser peuvent nécessiter plusieurs types

de contrôleurs (basés sur des règles, sur la théorie du contrôle continu ou discret).

5.1.1 Motivations

La plupart des systèmes adaptatifs sont constitués de nombreuses entités, par exemple, un bâtiment de plusieurs étages. L'adaptation d'un tel système ne peut pas être effectuée de façon monolithique avec une seule boucle. Cette boucle collecterait toutes les données et calculerait toutes les commandes. Cela dégraderait la réactivité du système. En effet, toutes les données du système doivent être lues avant de réagir à l'occurrence d'un événement. Par exemple, pour démarrer un extincteur dans une pièce lorsque de la fumée y est détectée, il faudrait lire toutes les données de l'ensemble des capteurs du bâtiment. De plus, cela causerait des problèmes en termes

de conception : cette boucle serait difficile à concevoir parce qu'il faut gérer une grande quantité de données, par exemple, les données de tous les capteurs des différents pièces du bâtiment de plusieurs étages. De plus, la conception du contrôleur d'une telle boucle requiert de spécifier les états de toutes entités du système. Il peut être nécessaire, dans le cas où le contrôleur est basé sur des règles ou est validé par la vérification formelle, de spécifier comment ces entités interagissent pour réaliser les objectifs du système. Dans ce cas, le développeur doit considérer tous les cas possibles afin d'écrire un ensemble de règles complet ou de modéliser l'ensemble des comportements du système.

de validation : cette boucle serait difficile à valider du fait du nombre élevé d'entités. Dans le cas où le contrôleur est conçu sous la forme d'un ensemble de règles écrites manuellement, le développeur doit détecter et résoudre des conflits entre les règles. Il doit pour ce faire, comparer l'ensemble des règles, modifier certaines règles en leur rajoutant des conditions et, si nécessaire, écrire de nouvelles règles. De plus, le développeur doit faire plusieurs itérations pour s'assurer que tous les conflits sont détectés et résolus. De même, lorsque le contrôleur est basé sur la théorie du contrôle discret (vérification formelle ou synthèse de contrôleurs discrets), le coût de validation est exponentiel en fonction du nombre de variables utilisées dans le modèle du système. Ce qui fait que, lorsque le nombre d'entités est élevé, la validation peut prendre du temps. Elle peut même échouer du fait de limitations de ressources en termes de CPU et/ou de RAM. La difficulté de valider le comportement de la boucle peut compromettre la fiabilité comportementale.

d'exécution : le coût de l'exécution de cette boucle serait élevé parce qu'elle lit de nombreuses données. Dans le cas où le contrôleur est une fonction de transitions, elle est invoquée dans une règle unique. Cette règle est ensuite exécutée par un moteur de règles qui crée un arbre d'inférence. Du fait du nombre élevé d'entités, la taille de l'arbre d'inférence est grande. Ce qui fait que le coût de son stockage et son parcours (charge en RAM et en CPU) est élevé. Lorsque le contrôleur est constitué d'un ensemble de règles, leur exécution peut être distribuée pour réduire la charge en CPU et en RAM.

Cependant, ces règles vont accéder au même instant à plusieurs données qui sont les mêmes, pour éviter les conflits, et en plus à travers le réseau. Cela peut dégrader la réactivité du système du fait du coût de communication.

La conception de boucles multiples est également motivée par le fait que certains systèmes doivent réaliser des objectifs qui requièrent différents types de contrôleurs. Par exemple, un système peut avoir des objectifs pouvant être réalisés sous la forme *si alors sinon*. Ce même système peut avoir d'autres objectifs, en termes de consignes à atteindre malgré la présence de perturbations extérieures, dont la réalisation requiert un contrôleur basé sur la théorie du contrôle continu. L'adaptation d'un tel système est effectuée par plusieurs boucles qui ont différents types de contrôleurs.

5.1.2 Avantages

Les boucles multiples permettent la structuration d'un système adaptatif, simplifient sa conception et son déploiement. Le système considéré est d'abord divisé en sous-systèmes. Ensuite, une boucle est conçue, comme présenté au chapitre 3, pour chaque sous-système. Un sous-système étant constitué d'un nombre limité d'entités, et non de l'ensemble des entités du système, le coût de conception d'une boucle d'un sous-système est plus faible que celui d'une boucle unique pour tout le système.

Par exemple, considérons un bâtiment B_1 qui est composé de quatre étages avec plusieurs pièces et couloirs chacun. Chaque pièce est équipée de capteurs et d'actionneurs. Les actionneurs de ce bâtiment doivent être contrôlés pour réaliser des objectifs relatifs au confort des occupants. Pour ce faire, le développeur peut concevoir une boucle pour chaque étage. Cela diminuerait, par rapport à une seule boucle pour tout le bâtiment, les coûts de conception, de validation et d'exécution. La raison est que la boucle d'un étage ne concerne que les entités de cet étage et ne lit que les données relatives à cet étage. Le développeur pourrait également concevoir une boucle pour chaque pièce. Une pièce étant constituée de moins d'entités qu'un étage, la conception d'une boucle de pièce permet de réduire les coûts de conception, de validation et d'exécution par rapport à ceux d'une boucle unique par étage.

Les boucles multiples permettent également de réutiliser des boucles ou des composants de boucles existants lors de la conception. En effet, une boucle qui est conçue pour l'adaptation d'un sous-système peut être réutilisée avec ou sans modification pour un ou plusieurs autres sous-systèmes. Dans l'exemple du bâtiment de plusieurs étages, une boucle de pièce peut être conçue puis réutilisée pour toutes les pièces qui sont équipées des mêmes types de capteurs et des mêmes types d'actionneurs.

Enfin, la conception de boucles multiples simplifie l'évolution des systèmes en permettant la modification des boucles des sous-systèmes sans affecter les autres boucles. Par exemple, lorsqu'un nouveau capteur et un nouvel actionneur sont rajoutés dans une pièce du bâtiment B_1, la boucle de cette pièce est modifiée et seules ces entités sont considérées lors de la modification. Dans le cas d'une seule boucle pour tout le bâtiment, toutes les entités de toutes les pièces devraient être considérées pour pouvoir intégrer dans la boucle le nouveau capteur et le nouvel actionneur.

5.2 Modes de composition de boucles

Le support intergiciel SICODAF permet de concevoir des boucles de déploiement ou des boucles applicatives qui sont en parallèle, coordonnées ou hiérarchiques.

Boucles en parallèle : chaque boucle effectue l'adaptation d'un sous-système indépendamment des autres boucles. Ce mode de composition est utilisé dans le cas où le système considéré peut être divisé en sous-systèmes indépendants. Des sous-systèmes sont indépendants lorsqu'ils n'ont pas d'entités en commun et lorsqu'un événement (resp. une action) survenu (resp. effectuée) dans un des sous-systèmes n'a pas d'effet dans un ou plusieurs autres sous-systèmes.

Boucles coordonnées : les boucles sont contrôlées par un coordinateur pour éviter les décisions conflictuelles et les violations d'objectifs. Ce mode de composition est utilisé lorsque les sous-systèmes ne sont pas indépendants.

Boucles hiérarchiques : les boucles sont conçues avec un ou plusieurs niveaux hiérarchiques. Les boucles du niveau hiérarchique le plus faible effectuent l'adaptation des sous-systèmes et sont contrôlées par des contrôleurs au niveau supérieur. Lorsque le niveau hiérarchique est égal à un, ce mode de composition est le même que le mode boucles coordonnées. Le fait de concevoir plusieurs niveaux hiérarchiques permet de contrôler les boucles d'un niveau en faisant abstraction des boucles qui ont un niveau hiérarchique inférieur.

5.3 Conception de boucles en parallèle

Le développeur divise le système considéré en sous-systèmes indépendants et conçoit une boucle pour chaque sous-système, comme illustré à la Figure 5.1. La division en sous-systèmes peut être guidée par la structure du système ou les objectifs qu'il doit réaliser. La conception d'une boucle pour un sous-système peut être effectuée de façon manuelle ou semi-automatique comme présentée au chapitre 3.

5.3.1 Avantages des boucles en parallèle

Les boucles en parallèle permettent de réduire les coûts de conception, de validation et d'exécution du système considéré grâce à sa division en sous-systèmes.

Réduction du coût de conception : le développeur conçoit la boucle de chaque sous-système en ne prenant en compte que les données (resp. les entités) de ce sous-système qui est un sous ensemble des données (resp. entités) du système.

Réduction du coût de validation : les sous-systèmes sont indépendants et donc leurs boucles sont validées séparément. Dans le cas où le contrôleur d'une boucle à valider est basé sur des règles, le développeur a moins de règles à analyser (les règles d'un sous-système et non celles de tout le système). De même, dans le cas où le contrôleur est basé sur la théorie du contrôle discret, le coût exponentiel de la vérification formelle, ou de la synthèse du contrôleur, est réduit car elle est effectuée sur un sous-ensemble des entités du système.

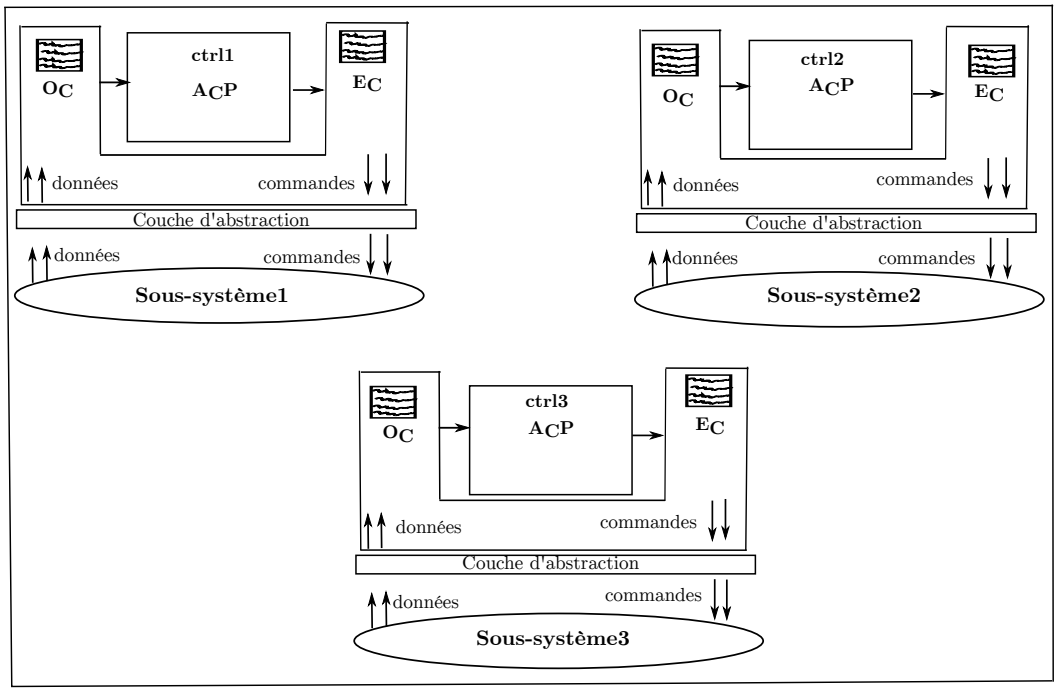

FIGURE 5.1 – Boucles en parallèle

Réduction du coût d'exécution : le contrôleur de chaque boucle est associé, pour son exécution, à une règle (plusieurs pour un contrôleur basé sur des règles). Les règles associées aux différents contrôleurs peuvent être exécutées à l'aide d'un moteur de règles. Pour limiter la taille de l'arbre d'inférence créé lors de l'exécution, les règles peuvent être exécutées à l'aide de plusieurs moteurs de règles. Dans ce cas, chaque moteur exécute un sous-ensemble de règles et crée un arbre d'inférence de taille limité. Ces moteurs peuvent être distribués sur plusieurs ressources de calcul pour réduire le coût d'exécution.

5.3.2 Limitations des boucles en parallèle

Les boucles en parallèle sont limitées par le fait que la plupart des systèmes ne peuvent pas être divisés en sous-systèmes qui sont tous indépendants. Les événements (resp. les actions) qui surviennent (resp. sont effectuées) dans certains sous-systèmes ont des effets sur d'autres sous-systèmes. Ces effets peuvent être indésirables, lorsqu'ils mènent à des décisions conflictuelles ou violent un objectif.

Par exemple, considérons le bâtiment B_1 décrit à la section 5.1.2. Un étage de ce bâtiment est constitué, comme illustré à la Figure 5.2, de deux couloirs séparés dans lesquels se trouvent différentes pièces. Si nous considérons un objectif qui consiste à limiter le niveau de bruit dans une pièce lorsqu'elle est occupée, nous pouvons faire l'hypothèse que les pièces qui sont dans des couloirs différents sont indépendantes (les couloirs sont séparés). Par contre, dans le même couloir, le niveau de bruit élevé d'une pièce, causé par exemple par des travaux, peut affecter les autre pièces et

violer l'objectif qui consiste à limiter le niveau de bruit lorsqu'elles sont occupées, ce qui doit être évité. Pour ce faire, les boucles des pièces du même couloir doivent être coordonnées, en fermant les fenêtres des autres pièces lorsqu'elles sont occupées et que le niveau de bruit dans une ou plusieurs pièces du même couloir est élevé.

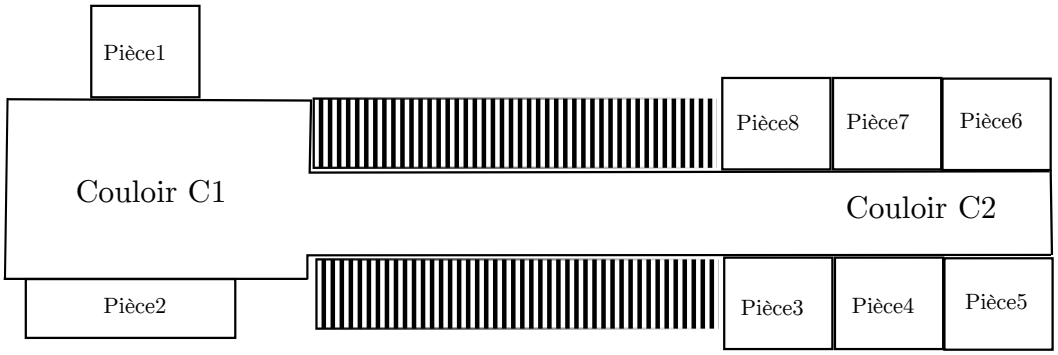

FIGURE 5.2 – Structure d'une étage de l'exemple du bâtiment B_1

5.4 Conception de boucles coordonnées

Le développeur identifie les boucles qui doivent être coordonnées et conçoit un coordinateur. Comme illustré à la Figure 5.3, le coordinateur collecte des données des contrôleurs et des sous-système et fournit des entrées aux contrôleurs. Le but est d'éviter les conflits entre les différentes boucles en empêchant le fait qu'elles aient au même instant des actions contradictoires sur une même entité ou qu'une boucle effectue une action pouvant violer un ou plusieurs objectifs d'un autre sous-système.

La conception de boucles coordonnées a des prérequis et le coordinateur peut être conçu sous la forme d'un ensemble de règles ou à l'aide du contrôle discret.

5.4.1 Prérequis de la conception d'un coordinateur

La conception de boucles coordonnées, pour un système, requiert que les contrôleurs des boucles qui doivent être coordonnées puissent recevoir des entrées de la part du coordinateur. Un tel contrôleur est dit contrôlable. Il possède une ou plusieurs entrées qui sont appelées variables de coordination. Une telle variable oblige, ou non selon sa valeur, le contrôleur à effectuer une action particulière sur une entité qu'il contrôle. Pour un contrôleur qui est contrôlable, le fait de l'obliger à effectuer une action (p. ex., fermer fenêtre) ne viole pas les objectifs qu'il doit réaliser parce qu'il pourra effectuer une action alternative si nécessaire (p. ex., démarrer le climatiseur).

Par exemple, considérons le bâtiment B_1 dans lequel une boucle a été conçue pour chaque pièce. Le contrôleur de la boucle d'une pièce contrôle les différents actionneurs (lampe, volet, fenêtre, climatiseur réversible, ventilation mécanique) pour assurer le confort des occupants et réduire la consommation d'énergie. Il réalise pour ce faire des objectifs relatifs à la luminosité, la température et au CO_2. Pour l'économie d'énergie, le contrôleur préfère ouvrir la fenêtre pour ventiler (resp. refroidir)

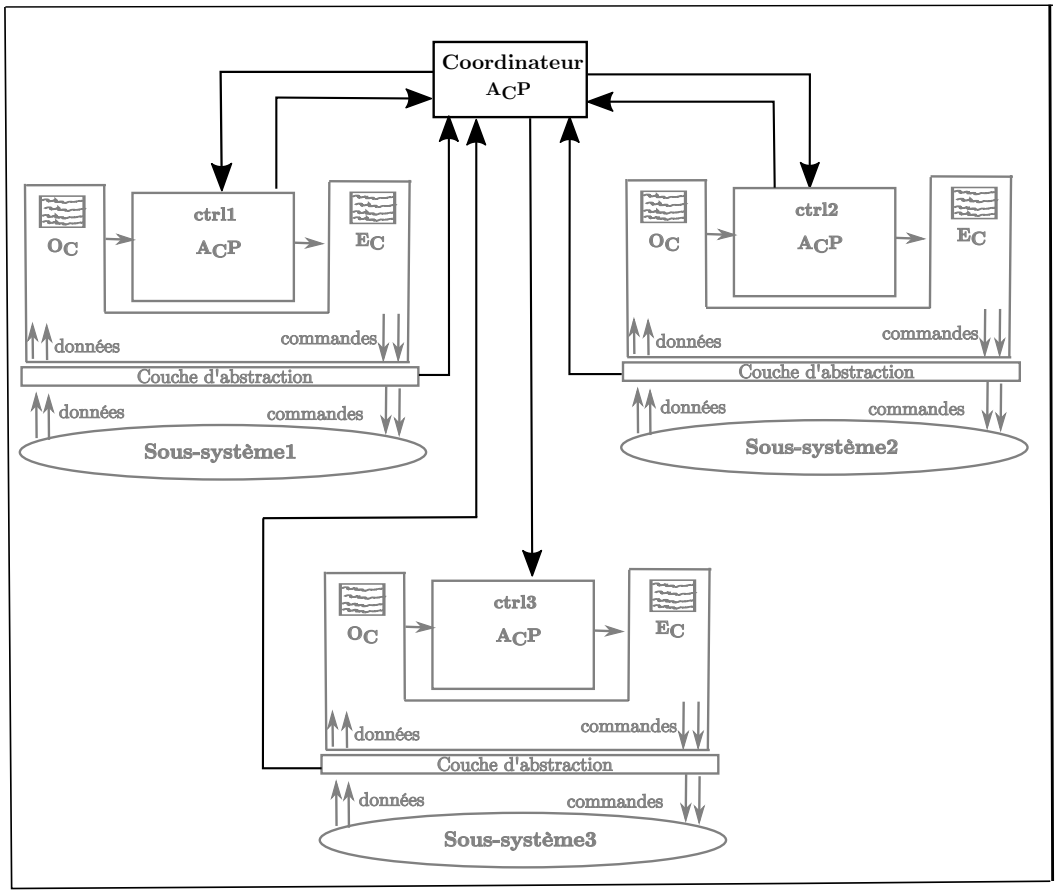

FIGURE 5.3 – Boucles coordonnées

la pièce lorsqu'une présence est détectée et que le CO_2 (resp. la température) est élevé. Le contrôleur de la boucle d'une pièce est contrôlable. Il possède une variable de coordination qui l'oblige à fermer la fenêtre lorsque la pièce est occupée et que le bruit dans une ou plusieurs autres pièces du même couloir est élevé (p. ex., du fait de travaux). Le fait d'obliger au contrôleur de fermer la fenêtre ne lui empêche pas de réaliser les objectifs relatifs à la température et au CO_2. La raison est que le contrôleur est conçu de sorte que s'il ne peut pas ouvrir la fenêtre pour ventiler (resp. refroidir) la pièce, il va utiliser la ventilation mécanique (resp. le climatiseur).

Un autre prérequis de la conception de boucles coordonnées est lié au fait que certains contrôleurs peuvent partager les mêmes entités. De tels contrôleurs peuvent avoir comme connaissance sur ces entités, des modèles qui représentent leurs états. Dans ce cas, les états des entités partagées doivent être les mêmes à tout instant pour tous les contrôleurs. Cela doit être effectué par le coordinateur, à l'aide des variables de coordination et des commandes calculées par les contrôleurs. Ces variables permettent, lorsqu'un contrôleur calcule une nouvelle commande pour une entité partagée avec d'autres contrôleurs, d'obliger les autres contrôleurs à changer l'état de cette entité pour qu'il soit le même pour tous les contrôleurs qui le partage.

5.4.2 Conception d'un coordinateur basé sur des règles

Le développeur écrit un ensemble de règles qui vérifient des conditions et donnent des valeurs aux variables de coordination des contrôleurs des boucles à coordonner.

Par exemple, considérons un étage du bâtiment B_1 (cf. Figure 5.2). Pour coordonner les pièces du couloir `C1`, afin d'empêcher que le bruit dans une pièce affecte l'autre lorsqu'elle est occupée, le développeur doit écrire deux règles : `R1p1` et `R1p2`. Le Listing 5.1 présente la règle `R1p1`. Cette règle vérifie si une présence est détectée dans `Piece1` et si le niveau de bruit dans `Piece2` est élevé puis donne la valeur *vrai* à la variable de coordination du contrôleur de la boucle de `Piece1` pour qu'il ferme la fenêtre. La règle `R1p2` vérifie la présence dans `Piece2` et le bruit dans `Piece1` puis donne la valeur *vrai* à la variable de coordination du contrôleur de la boucle de `Piece2` pour éviter qu'elle ne soit affectée par le bruit. Le développeur doit également écrire deux autres règles qui changent les valeurs des variables de coordination des contrôleurs à `faux`. Une telle règle spécifie à un contrôleur qu'il n'est plus obligé de fermer la fenêtre. Par exemple, le Listing 5.2 présente la règle qui change la variable de coordination du contrôleur de la boucle de `Piece1` à `faux`. Cette règle vérifie si une présence n'est pas détectée dans `Piece1` ou le bruit dans `Piece2` n'est pas élevé.

```
Regle R1p1

Surveillance
  Donnees
    EnOcean.Sensors.lire("presence_P1", presence_P1_val)
    TelosB.Sensors.lire("bruit_P2", bruit_P2_val)
  Conditions
    presence_P1_val = vrai et bruit_P2 = eleve

Mise a jour transactionelle
  Actions
    Conditions
      presence_P1_val = vrai et bruit_P2 = eleve
    Commandes
      ferme_fenetre_P1 = vrai
```

Listing 5.1 – Exemple de règle pour la coordination de boucles

```
Regle R2p1

Surveillance
  Donnees
   EnOcean.Sensors.lire("presence_P1", presence_P1_val)
   TelosB.Sensors.lire("bruit_P2", bruit_P2_val)
  Conditions
   presence_P1_val = faux ou bruit_P2 != eleve

Mise a jour transactionelle
  Actions
    Conditions
      presence_P1_val = faux ou bruit_P2 != eleve
    Commandes
      ferme_fenetre_P1 = faux
```

Listing 5.2 – Exemple de règle pour la coordination de boucles

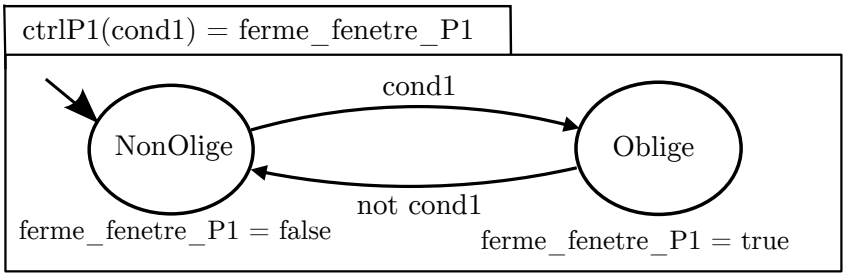

FIGURE 5.4 – Contrôle du contrôleur de la boucle de `Piece1`

5.4.3 Conception d'un coordinateur basé sur le contrôle discret

Le coordinateur peut être conçu de façon manuelle puis validé par la vérification formelle. Il peut aussi être conçu en utilisant la synthèse de contrôleurs discrets.

5.4.3.1 Coordinateur validé par la vérification formelle

Le développeur modélise d'abord le contrôleur de chacune des boucles à coordonner, sous la forme d'un système de transitions. Lors de la modélisation d'un contrôleur, le développeur doit spécifier les comportements qui sont pertinents pour la coordination. Ces comportements sont ceux qui peuvent être contrôlés pour éviter les conflits et violations d'objectifs. Ensuite, le développeur modélise la logique de coordination et vérifie le modèle des contrôleurs coordonnés (la composition des modèles des contrôleurs et de la logique de coordination). Le but est de vérifier que les contrôleurs ne prennent pas de décisions conflictuelles et ne violent pas d'objectifs. Si c'est le cas, la logique de coordination est valide et peut être mise en œuvre.

Par exemple, considérons le couloir $C1$ du bâtiment $B1$ (cf. Figure 5.2) et l'objectif de coordination qui consiste à éviter que le bruit dans une pièce, du fait de travaux, n'affecte l'autre pièce lorsqu'elle est occupée. Pour concevoir et valider le coordinateur des boucles de ces pièces, trois automates `ctrlP1`, `ctrlP2` et `coordinationC1` sont définis. `ctrlP1` (resp. `ctrlP2`) modélise le comportement du contrôleur de la boucle `Piece1` (resp. `Piece2`) par rapport à la fenêtre. Le comportement du contrôleur par rapport aux autres actionneurs n'est pas pertinent pour l'objectif de coordination qui consiste à éviter le bruit provenant d'une autre pièce. `coordinationC1` modélise la logique de coordination des deux contrôleurs.

La Figure 5.4 montre l'automate `ctrlP1`. Cet automate a deux états et deux transitions qui spécifient que `ctrlP1` peut selon une condition extérieure, `cond1`, être obligé (`Oblige`) ou non (`NonOblige`) de fermer la fenêtre de `Piece1`. La Figure 5.5 montre l'automate `ctrlP2`. Cet automate spécifie également que `ctrlP2` peut selon une condition extérieure `cond2` être obligé ou non de fermer la fenêtre de `Piece2`. La Figure 5.6 montre l'automate qui modélise la logique de coordination. Cet automate définit est la composition des automates des contrôleurs et de deux équations. Ces équations spécifient quand est ce que les contrôleurs sont obligés ou non de fermer les fenêtres qu'ils contrôlent, en donnant des valeurs à `cond1` et `cond2` en fonction

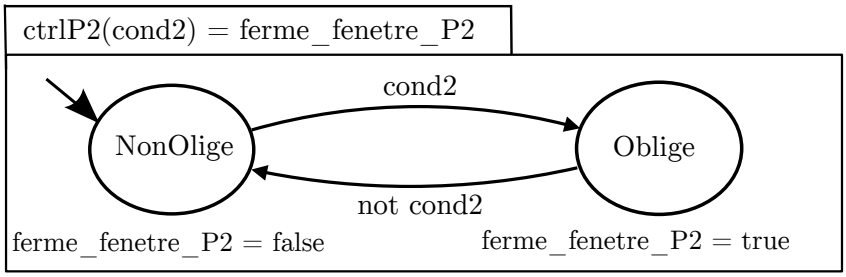

FIGURE 5.5 – Contrôle du contrôleur de la boucle de `Piece2`

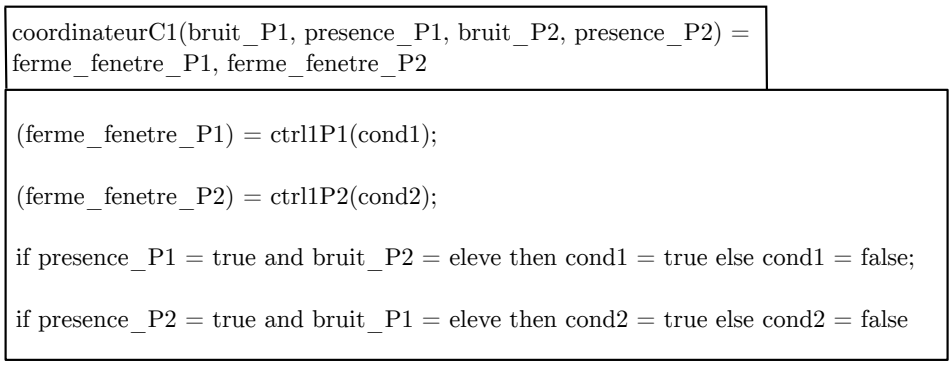

FIGURE 5.6 – Coordinateur du couloir `C1` par vérification formelle

du niveau de bruit et de la présence de chacune des pièces. La composition des automates et la logique de coordination est vérifié à l'aide d'un outil de vérification. Pour ce faire, deux propriétés *Prop1* et *Prop2* sont définies

— *Prop1* : `presence_P1 and bruit_P2 = eleve` ⇒ `ferme_fenetre_P1`
— *Prop2* : `presence_P2 and bruit_P1 = eleve` ⇒ `ferme_fenetre_P2`

Ces propriétés permettent de vérifier si lorsqu'une pièce est occupée et qu'il y a du bruit, dans l'autre pièce, le contrôleur de la pièce sera obligé de fermer la fenêtre.

5.4.3.2 Coordinateur basé sur la synthèse de contrôleurs discrets

Le développeur modélise les contrôleurs des boucles, sous la forme de systèmes de transitions, et définit un *contrat*. Ce *contrat* spécifie les propriétés qui doivent être valides pour la coordination et les points de contrôlabilité. Cela permet la génération de la logique de coordination à l'aide d'un outil de synthèse de contrôleurs discrets.

Par exemple, considérons le couloir $C1$ du bâtiment $B1$. Le modèle du contrôleur de la boucle de `Piece1` (resp. `Piece2`) est celui de la Figure 5.4 (resp. 5.5). La Figure 5.7 présente le *contrat* qui est défini pour la coordination des contrôleurs. La propriété qui doit être valide est : lorsqu'une présence est détectée dans une pièce et que le bruit dans l'autre pièce est élevé, le contrôleur doit être obligé de fermer la fenêtre (*prop1* et *prop2*). Les points de contrôlabilité sont les conditions sous lesquelles les contrôleurs sont obligés, ou pas, de fermer les fenêtres (`cond1` et `cond2`). La synthèse de contrôleurs discrets est effectuée afin de générer le coordinateur.

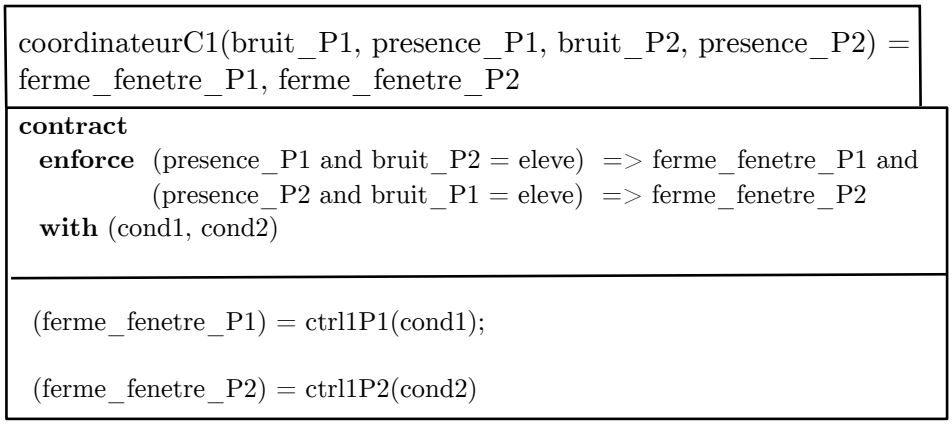

FIGURE 5.7 – Coordinateur du couloir `C1` par synthèse de contrôleurs discrets

5.4.4 Limitations des boucles coordonnées

Les boucles coordonnées sont limitées par le fait que lorsque tous les sous-systèmes interagissent , s un coordinateur unique doit être conçu. Un tel coordinateur peut être difficile à concevoir lorsque le nombre de sous-systèmes considérés est élevé.

Par exemple, considérons un étage du bâtiment B_1 (cf. Figure 5.2). Les boucles des pièces de chaque couloir sont coordonnées par un coordinateur pour éviter que le bruit d'une pièce affecte une autre pièce qui est occupée. Considérons un nouvel objectif qui consiste à éviter le fait qu'il y ait du courant d'air. Par rapport à ce nouvel objectif, les pièces des deux couloirs ne sont pas indépendantes. En effet, le fait d'ouvrir une fenêtre dans au moins une pièce de chaque couloir peut introduire un courant d'air et violer l'objectif. Pour éviter la violation de cet objectif, les boucles des pièces des deux couloirs doivent être coordonnées. Cela est effectué par la conception d'un nouveau coordinateur qui contrôle les boucles de toutes les pièces pour réaliser les objectifs considérés. Dans ce cas, les coordinateurs des pièces de chaque couloir ne sont pas réutilisés et le coût de conception du nouveau coordinateur peut être élevé du fait qu'il coordonne les boucles de toutes les pièces de l'étage. Une solution à cette limitation est la conception de boucles hiérarchiques.

5.5 Conception de boucles hiérarchiques

Le développeur conçoit des boucles avec plusieurs niveaux hiérarchiques. Comme illustré à la Figure 5.8, les boucles du niveau hiérarchique le plus faible contrôlent les sous-systèmes et sont contrôlées par des contrôleurs qui peuvent eux aussi être contrôlés. La conception de boucles hiérarchiques permet de contrôler des contrôleurs de sous-systèmes qui interagissent. Par exemple, dans un étage du bâtiment B_1, pour éviter qu'il y ait du courant d'air, les coordinateurs pour le bruit, des boucles des pièces, des couloirs sont contrôlés. Cela permet la réutilisation des coordinateurs.

La conception de boucles hiérarchiques requiert que les contrôleurs des boucles

puissent être contrôlés par des contrôleurs pouvant être contrôlés. Ces contrôleurs possèdent des variables de coordination et peuvent être conçus en utilisant le langage de règles, fourni par l'intergiciel à base de tuples, ou la théorie du contrôle discret.

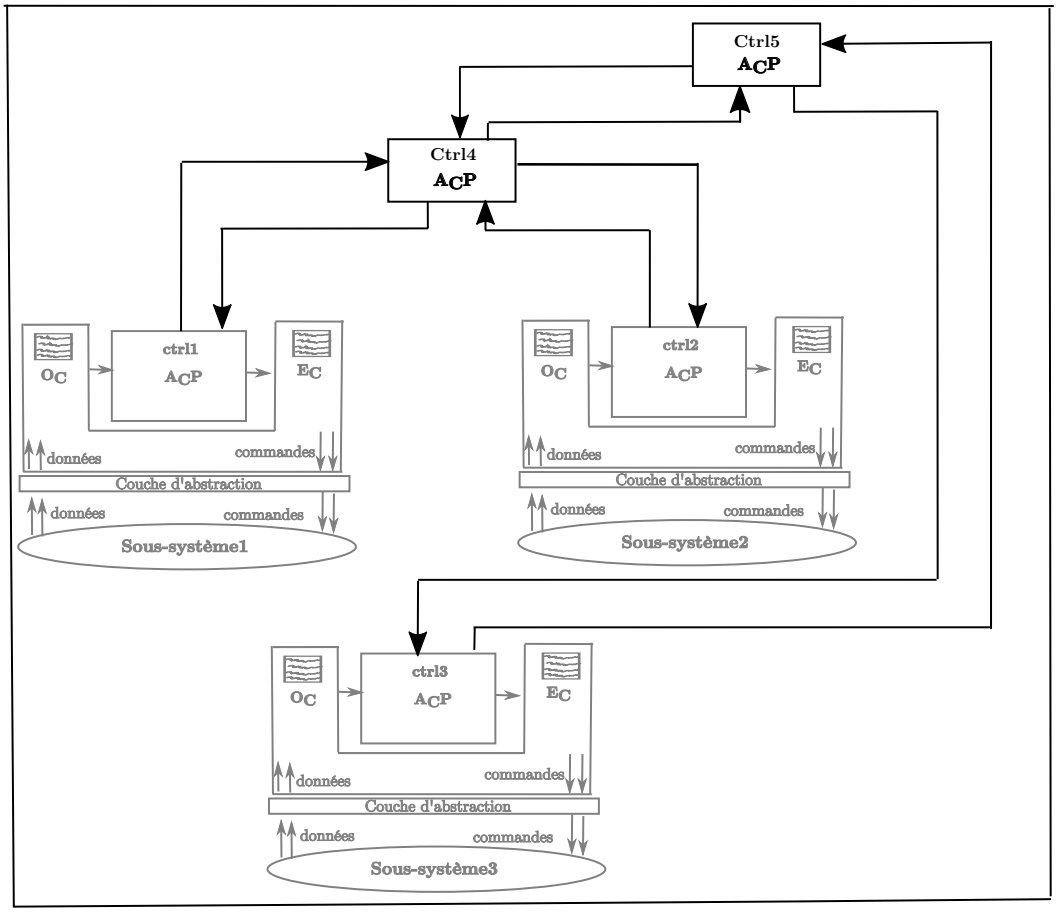

FIGURE 5.8 – Boucles hiérarchiques

5.5.1 Conception à l'aide du langage de règles

Chaque contrôleur est conçu sous la forme d'un ensemble de règles. Les règles d'un contrôleur du niveau hiérarchique le plus faible, par exemple `ctrl1` de la Figure 5.8 vérifient des conditions sur un sous-système et exécutent des commandes. Ces règles peuvent recevoir, pour le calcul des commandes, des entrées de la part d'autres règles qui correspondent aux contrôleurs du niveau hiérarchique supérieur.

5.5.2 Conception à l'aide de la théorie du contrôle discret

Les contrôleurs des boucles hiérarchiques peuvent validés par la vérification formelle. Ils peuvent également être conçus à l'aide de la synthèse de contrôleurs discrets.

5.5.2.1 Validation à l'aide de la vérification formelle

Les comportements des contrôleurs sont d'abord modélisés sous la forme d'un système de transitions hiérarchique pouvant être, par exemple, un réseau de Petri. Ensuite, le système de transitions est vérifié pour détecter des conflits et des violations d'objectifs. La nature hiérarchique du système de transitions permet de structurer le système et de réduire les coûts de conception des contrôleurs. Le système de transitions hiérarchique permet aussi d'effectuer la vérification formelle de façon modulaire, comme illustré dans [2, 53]. Cela réduit les coûts de validation.

5.5.2.2 Conception à l'aide de la synthèse de contrôleurs discrets

La synthèse de contrôleurs discrets modulaire est utilisée pour générer les contrôleurs des boucles. Les contrôleurs du niveau hiérarchique le plus faible sont d'abord spécifiés sous la forme d'un modèle comportemental et d'un *contrat*. Ensuite, ces spécifications sont réutilisées pour les contrôleurs de niveau supérieur. Enfin, la synthèse de contrôleurs discrets modulaire est effectuée sur les spécifications des contrôleurs.

Par exemple, considérons sur la Figure 5.8, la boucle hiérarchique qui est constituée des contrôleurs `ctrl1`, `ctrl2` et `ctrl4`. Les contrôleurs `ctrl1` et `ctrl2` sont d'abord spécifiés comme présenté à la Figure 5.9 pour le contrôleur `ctrl1`. Ensuite, ces spécifications sont réutilisées pour `ctrl4`, comme présenté à la Figure 5.10.

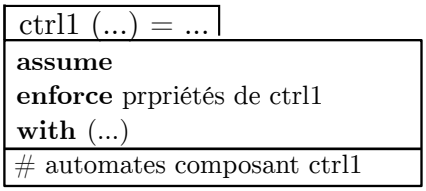

FIGURE 5.9 – Spécification du contrôleur `ctrl1`

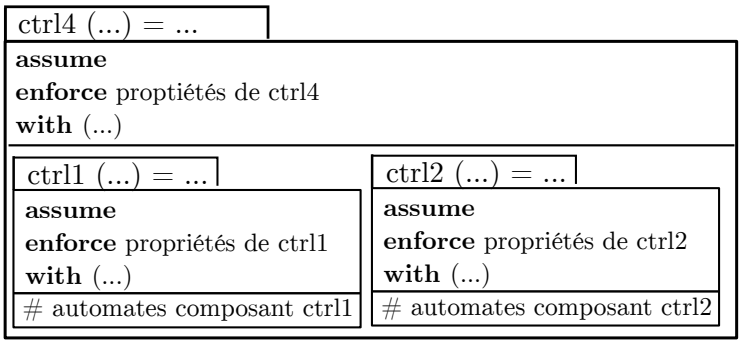

FIGURE 5.10 – Spécification du contrôleur `ctrl4`

Le *contrat* dans la spécification d'un contrôleur doit définir suffisamment de propriétés, d'hypothèses et de points de contrôlabilité pour que le contrôleur puisse être contrôlé au niveau supérieur. En effet, le *contrat* est utilisé pour abstraire le comportement du contrôleur lors de la synthèse de contrôleurs discrets modulaire.

Le fait d'effectuer la synthèse de contrôleurs discrets modulaire permet de générer une fonction de transitions pour chaque contrôleur. Elle permet également de réduire les coûts de synthèse. La synthèse est effectuée sur chaque sous-système (le modèle comportemental de chaque contrôleur) et non sur l'ensemble du système.

Parmi les fonctions de transitions générées, figure une fonction principale. Elle est en charge d'exécuter les autres fonctions de transitions. Pour permettre l'exécution des contrôleurs, la fonction de transitions principale est invoquée dans une règle.

Cette règle collecte les données et invoque la fonction de transitions principale qui exécute les autres fonctions de transitions. Cela est limitant car une seule règle est utilisée pour exécuter plusieurs fonctions de transitions. Une solution serait d'invoquer chaque fonction de transitions dans une règle et de gérer leurs interactions (une sortie d'une fonction de transitions peut être une entrée d'une autre). La gestion des interactions peut être validée par la vérification formelle comme dans [34].

5.6 Conclusion

Ce chapitre a présenté la conception de boucles multiples à l'aide du support intergiciel SICODAF. Des boucles multiples sont conçues pour un système qui est constitué d'un nombre élevé d'entités ou qui requiert différents types de contrôleurs. Pour ce faire, le système est d'abord divisé en plusieurs sous-systèmes et une boucle est conçue pour chacun d'eux. La conception de boucles multiples permet de structurer un système et de réduire les coûts liés à sa conception, sa validation et son exécution. La réduction des coûts est rendue possible par la décomposition du système en sous-systèmes. La conception de boucle multiples permet également la réutilisation de boucles existantes pour un autre système. Enfin, elle permet l'évolution d'un système par la modification d'une ou de plusieurs boucles de ses sous-systèmes.

Les boucles multiples conçues à l'aide de SICODAF peuvent être composées suivant trois modes (parallèle, coordonnée et hiérarchique). Le mode de composition est choisi en fonction des interactions entre les sous-systèmes considérés et le nombre d'entités d'un sous-système. Le mode parallèle est utilisé lorsque les sous-systèmes sont indépendants. Le mode coordonné est utilisé lorsque certains sous-systèmes interagissent. Le mode hiérarchique peut être utilisé lorsque le système considéré possède une structure hiérarchique ou lorsque plusieurs sous-systèmes interagissent. Les boucles multiples peuvent être conçues en utilisant le langage de règle de l'intergiciel à base de tuples, la vérification formelle ou la synthèse de contrôleurs discrets.

Chapitre 6

Mise en œuvre et Études de cas

Sommaire

6.1 Implémentation 111
6.1.1 Combinaison de LINC et Heptagon/BZR 112
6.1.1.1 Invocation de la fonction *step* 112
6.1.1.2 Encapsulation de la fonction *Step* 115
6.1.2 Implémentation de boucles de déploiement 116
6.1.2.1 Implémentation de la couche d'abstraction 116
6.1.2.2 Langage de description textuel 118
6.1.2.3 Générateur de règles exécutant une fonction de transitions 118
6.1.3 Implémentation de boucles applicatives 119
6.1.3.1 Générateur de couches d'abstraction 119
6.1.3.2 Générateur de règles *templates* 119
6.1.3.3 Générateur de règles instances 120
6.2 Études de cas 120
6.2.1 Déploiement d'un système de traitement de données 120
6.2.1.1 Description des entités du système 120
6.2.1.2 Description des objectifs du système 121
6.2.1.3 Mise en œuvre d'une boucle de déploiement 121
6.2.1.4 Comportement de la boucle 125
6.2.2 Conception d'une pièce de bureau intelligente 127
6.2.2.1 Description des capteurs et des actionneurs 127
6.2.2.2 Description des objectifs 128
6.2.2.3 Mise en œuvre d'une boucle applicative 129
6.3 Conclusion 138

Ce chapitre présente la mise en œuvre du support intergiciel SICODAF et sa validation expérimentale. Il décrit d'abord comment SICODAF a été implémenté à l'aide d'outils concrets. Ensuite, il présente deux études de cas pour la validation.

6.1 Implémentation

Le support intergiciel SICODAF a été implémenté à l'aide du langage basé sur des systèmes de transitions Heptagon/BZR, de l'intergiciel transactionnel LINC et de l'environnement d'abstraction de technologies de communication PUTUTU.

Cette implémentation a été effectuée comme suit : LINC a été d'abord combiné à Heptagon/BZR, pour garantir à la fois la fiabilité comportementale et la fiabilité d'exécution des systèmes. Ensuite, deux types de boucles ont été implémentés : boucle de déploiement et boucle applicative dans le contexte du bâtiment intelligent.

6.1.1 Combinaison de LINC et Heptagon/BZR

Cette section présente comment LINC a été combiné à Heptagon/BZR (H/BZR). Le contrôleur de la boucle du système considéré est d'abord mis en œuvre sous la forme d'un programme H/BZR (cf. section 2.3 du chapitre 2). Ensuite, le programme est compilé pour effectuer la synthèse de contrôleurs discrets et générer du code C. Ce code contient (i) une fonction de transitions appelée *step*, (ii) une variable appelée *memory* qui contient l'état de l'automate modélisant le système et (iii) une fonction d'initialisation appelée *reset* qui initialise la variable *memory*. La fonction *step* est encapsulée dans un sac (mémoire associative de LINC), appelé *Step* contenu dans un objet LINC, puis invoquée dans une règle LINC, pour son exécution [126].

6.1.1.1 Invocation de la fonction *step*

L'invocation du *step*, dans une règle LINC pour son exécution, est effectuée sur la base des fondements de H/BZR et de LINC. Ces fondements sont rappelés ci-après.

Rappel des fondements de LINC et H/BZR Le *step* prend en entrée les données collectées, calcule les commandes à exécuter et met à jour l'état de l'automate global qui modélise le système. Une exécution du *step* correspond à une réaction du système et doit être effectuée à chaque fois que des changements surviennent. H/BZR étant un langage synchrone, l'exécution du *step* requiert de garantir l'hypothèse synchrone qui stipule qu'une réaction du système est plus rapide que sa dynamique et celle de son environnement : le système n'évolue pas durant l'exécution du *step*.

Une règle LINC, quant à elle, est constituée de deux parties : une *précondition* et une *performance*. Elle est déclenchée à chaque fois qu'une nouvelle instance d'une donnée qu'elle lit est produite. La *précondition* collecte des données sur le système à l'aide des capteurs, sous la forme de tuples pouvant être partiellement instanciés. La *performance* vérifie si les données collectées sont toujours valides, exécute une ou plusieurs commandes, à l'aide des actionneurs, et met à jour l'état logique (une représentation de l'état réel) du système, dans une transaction. Les tuples manipulés dans la *performance* doivent être complètement instanciés, lors de la *précondition*.

Invocation du *step* Le *step* met à jour l'état de l'automate du système et la *performance* d'une règle LINC est utilisée pour mettre à jour l'état logique du système. Par conséquent, le *step* devrait être invoqué dans la *performance* d'une règle LINC, en effectuant une opération *put* sur le sac *Step*, de l'objet *HBZR*, qui encapsule la fonction *step*. Dans ce cas, l'opération *put* prendrait en paramètre un

tuple qui contient des variables instanciées et des variables non instanciées. Les variables instanciées correspondent aux valeurs des données collectées. Les variables non instanciées, quant à elles, sont utilisées pour stocker les commandes qui seront calculées par le *step* une fois exécuté. Cela n'est pas conforme avec la logique de LINC qui stipule que tous les tuples qui sont manipulés dans la *performance* d'une règle doivent être complètement instanciés, lors de la *précondition* de la même règle.

Pour résoudre ce problème, la solution proposée est la suivante. Le *step* est d'abord invoqué dans la *précondition* de la règle sans changer l'état de l'automate. L'objectif est de calculer les commandes et d'instancier les variables qui sont utilisées pour les stocker. Ensuite, le *step* est invoqué une deuxième fois dans la *performance* de la règle pour changer l'état de l'automate après avoir envoyé les commandes, dans une transaction. Dans ce cas, il faut veiller à ce que les deux invocations du *step* donnent le même résultat. En effet, si l'état actuel de l'automate change avant l'exécution de la *performance* (du fait de l'exécution d'une autre instance de la même performance), la deuxième invocation du *step* donnera un autre résultat, différent de celui obtenu dans la *précondition*. Dans cas, les commandes calculées ne sont plus valides et devront être recalculées avant d'être envoyées. Il faut aussi veiller à ce que le *step* qui change l'état de l'automate soit exécuté par une seule instance de la *performance*, sur des entrées qui sont valides. Cela garantit l'hypothèse synchrone.

Cette solution a été implémentée comme suit. La *précondition* de la règle effectue d'abord une opération *rd* d'un tuple dans un sac pour pour collecter les données sur le système. Ensuite, la *précondition* stocke les données lues dans une chaîne de caractères appelée *entrees* et effectue une opération *rd* sur le sac *Step* de l'objet *HBZR*. Ce *rd* invoque le *step* avec la valeur instanciée de la variable *entrees* pour calculer les commandes à exécuter sur le système. Dans cette invocation, le *step* ne change pas l'état de l'automate modélisant le système. Il retourne l'état courant de l'automate et les commandes à envoyer respectivement, dans deux variables : *etatCourant* et *cmds*. Ces variables sont instanciées et utilisées dans la *performance*.

La *performance* de la règle, dans une transaction, vérifie d'abord si les données collectées sont toujours valides en effectuant des opérations *rd*. Ensuite, elle effectue un *rd* du tuple *("allowed")* dans le sac *StepAccess* de l'objet *HBZR*. Cela verrouille le tuple *("allowed")* durant l'exécution de la transaction qui est en cours. Un autre instance de la même transaction voulant utiliser ce tuple doit attendre jusqu'à ce qu'il soit déverrouillé, à la fin de l'exécution de l'instance qui est en cours. Cela garantit l'hypothèse synchrone qui stipule que l'exécution de la règle invoquant le *step* doit être plus rapide que l'évolution du système (production de nouvelles données). En effet, le *step* qui change l'état de l'automate n'est exécuté, à un instant donné, que par une seule instance de la transaction, avec les données qui ont été collectées sur le système dans la *précondition* de la règle. Si de nouvelles données sont produites avant la fin de l'exécution de la règle, l'instance de transaction qui est en cours avorte (la vérification de la validité des anciennes données échoue) et une autre instance de la même transaction est déclenchée avec les nouvelles données.

Après le *rd* du tuple *("allowed")*, la transaction envoie les commandes calculées aux entités du système en effectuant des opérations *put*. Enfin, la transaction effectue

une opération *put* du tuple *(entrees,etatCourant,"")* dans le sac *Step*. Ce *put* vérifie d'abord si l'état courant de l'automate modélisant le système est toujours égal à *etatCourant*. Ensuite, il invoque le *step*, avec comme paramètre, la valeur de la variable *entrees*. Lors de cette invocation, le *step* renvoie les mêmes commandes, que celles qui ont été calculées dans la *précondition*, et met à jour l'état de l'automate. La transaction échoue si au moins une de ses opérations échoue. Sinon, elle réussit, les commandes sont envoyées et le *step* est exécuté pour changer l'état de l'automate. Le fait de changer l'état de l'automate à la fin de la transaction permet de ne pas annuler l'exécution du *step* lorsqu'une des opérations précédentes échoue. En effet, le *step* n'est exécuté que si toutes les opérations qui le précèdent sont réussies.

Exemple d'invocation du *step* Considérons le programme H/BZR qui est présenté à la Figure 2.8 du chapitre 2. Ce programme éteint toutes les ressources de calcul d'une pièce, dans un bâtiment de deux pièces, lorsqu'une présence n'y est pas détectée. Le *step* généré lors de la compilation de ce programme prend en entrée deux variables qui correspondent aux valeurs mesurées par le capteur de présence de chaque pièce. Ensuite, il retourne les commandes à envoyer aux différentes ressources de calcul du bâtiment et met à jour l'état des automates qui les modélisent.

```
1  ["ModBus","Sensors"].rd("pr1",pr1_val)&
   ["TelosB","Sensors"].rd("pr2",pr2_val)&
3  INLINE_COMPUTE: entrees = "(%s,%s)" %(pr1_val,pr2_val)&
   ["HBZR","Step"].rd(entrees,etatCourant,cmds)&
5  cmd_rsc12,cmd_rsc13,cmd_rsc14,cmd_rsc15,cmd_rsc_16,cmd_rsc17 = eval(cmds)
   ::
7  {
    ["ModBus","Sensors"].rd("pr1",pr1_val);
9   ["TelosB","Sensors"].rd("pr2",pr2_val);
    ["HBZR","StepAccess"].rd("allowed");
11  ["RscCalc","Commande"].put("rsc12",cmd_rsc12);
    ["RscCalc","Commande"].put("rsc13",cmd_rsc13);
13  ["RscCalc","Commande"].put("rsc14",cmd_rsc14);
    ["RscCalc","Commande"].put("rsc15",cmd_rsc15);
15  ["RscCalc","Commande"].put("rsc16",cmd_rsc16);
    ["RscCalc","Commande"].put("rsc17",cmd_rsc17);
17  ["HBZR","Step"].put(entrees,etatCourant,"")
   }.
```

Listing 6.1 – Exemple de règle LINC invoquant un *step*

Le Listing 6.1 présente la règle LINC qui exécute ce *step*. La *précondition* de la règle lit d'abord les valeurs mesurées par les deux capteurs de présence dans les variables `pr1_val` et `pr2_val` (lignes 1 et 2). Ensuite, elle les stocke sous la forme d'une chaîne de caractères dans une variable, appelée `entrees`, en utilisant l'opérateur `INLINE_COMPUTE` (ligne 3). Cet opérateur permet d'effectuer, dans la *précondition* d'une règle LINC, des commandes Python. Ensuite, la *précondition* de la règle effectue une opération *rd* sur le sac `Step` de l'objet `HBZR` (ligne 4) pour calculer les commandes et instancier la variable utilisée pour les stocker (`cmds`). Une fois instanciée, cette variable est utilisée dans la *performance* de la règle.

La *performance* est constituée d'une seule transaction. Cette transaction vérifie d'abord si les valeurs mesurées par les capteurs de présence n'ont pas changé. En-

suite, elle effectue une opération *rd*, du tuple `("allowed")`, dans le sac `StepAccess` de l'objet `HBZR` (ligne 10) pour garantir l'hypothèse synchrone. Ensuite, la transaction envoie les commandes calculées aux ressources de calcul (lignes 11 à 16). Enfin, elle effectue une opération *put* du tuple `(entrees,etatCourant,"")` dans le sac `Step` pour changer les états des automates modélisant les ressources de calcul.

Les actions asynchrones (p. ex. , ouverture d'une porte, d'une fenêtre ou d'un volet) dont l'exécution prennent plus de temps que l'exécution du *step* peuvent être modélisées à l'aide d'états transitoires. Ces états permettent d'attendre la fin de l'exécution de l'action, qui sera notifiée par un capteur. Cette notification va déclencher le *step* et ce dernier mettra à jour l'état de l'entité, comme illustré dans [18].

6.1.1.2 Encapsulation de la fonction *Step*

Cette section explique comment les opérations *rd* et *put* du *Step* de l'objet *HBZR* ont été implémentées. Ces opérations permettent respectivement, d'exécuter le *step* d'un programme H/BZR sans modifier l'état de l'automate ou en le mettant en jour.

Le code C associé au programme H/BZR, du système, est d'abord utilisé pour générer une librairie Python appelée *BZRLINC*. La raison est que LINC est écrit en Python. Ensuite, un objet LINC de type *H/BZR* utilisant ce module est défini.

BZRLINC Il permet d'exécuter le code C en Python et contient les fonctions :

- ***get_state()*** : cette fonction ne prend pas de paramètres. Elle renvoie sous la forme d'une chaîne de caractères, la valeur de la variable *memory*, du code C, qui contient l'état courant de l'automate modélisant le système considéré ;
- ***do_reset()*** : cette fonction ne prend pas de paramètres. Elle appelle la fonction *reset*, du code C, qui initialise la valeur de la variable *memory* ;
- ***do_pre_step()*** : cette fonction prend en entrée les données qui sont collectées sur le système. Lorsqu'elle est exécutée, cette fonction copie, dans un premier temps, la valeur courante de la variable *memory* dans une autre variable appelée *memoryCopy*. Ensuite elle exécute la fonction *step*, du code C, en utilisant les données collectées et la variable *memoryCopy*. Cela permet de calculer les commandes sans changer l'état de l'automate. Enfin, cette fonction retourne les commandes sous la forme d'une chaîne de caractères ;
- ***do_step()*** : cette fonction prend en entrée les données collectées et exécute le *step*, du code C, sur la variable *memory*. Cela permet de calculer les commandes et de mettre à jour l'état de l'automate modélisant le système.

Objet HBZR Le code de cet objet est une classe Python. Cette classe contient quatre méthodes qui correspondent aux fonctions du module *BZRLINC*. Elle contient également deux attributs *Step* et *StepAccess* (mémoires associatives) qui correspondent à des objets de deux autres classes : *bag* et *Step_bag*. Chacune de ces classes contient des méthodes parmi lesquelles figurent *put* et *rd*. Pour la classe *bag* ces méthodes permettent, respectivement, de stocker des tuples en mémoire et de

lire leurs valeurs. Pour la classe *Step_ bag*, ces méthodes sont redéfinies pour pouvoir exécuter la fonction *step*. La redéfinition de ces méthodes est effectuée comme suit :

- ***rd*** : il calcule les commandes sans changer l'état de l'automate. Ce *rd* prend en paramètre un tuple dont le motif est égal à *(entrees, etatCourant, commandes)*. Dans ce motif, *entree* est une variables instanciée tandis que *etatCourant* et *commandes* sont non instanciées et leurs valeurs seront retournées par le *rd*. Pour ce faire, le *rd* appelle d'abord la méthode *get_ state* de l'objet *HBZR* et stocke le résultat dans *etatCourant*. Ensuite, il appelle la méthode *do_ pre_ step()* et stocke le résultat dans *commandes* et retourne le tuple ;
- ***put*** : il change l'état de l'automate. Ce *put* prend en paramètre un tuple de motif *(entrees, etatCourant, "")* où *entrees* et *etatCourant* sont toutes les deux des variables instanciées. Le *put* vérifie d'abord, à l'aide de la méthode *get_ state()* que la valeur de la variable *memory* est toujours égale à *etatCourant*. Si c'est le cas, le *put* appelle la fonction *step*, du code C, en utilisant les entrées et la variables *memory*. Cela permet de changer l'état de l'automate.

6.1.2 Implémentation de boucles de déploiement

L'implémentation effectuée est relative à la couche d'abstraction, au langage de description textuel et au générateur de règles exécutant une fonction de transitions.

6.1.2.1 Implémentation de la couche d'abstraction

La couche d'abstraction, pour les boucles de déploiement, a été mis en œuvre par la création de quatre types d'objets LINC : *Materiel*, *Logiciel*, *Objet* et *Regle*.

Objet de type *Matériel* Il contient trois sacs, *Commande*, *Configuration* et *Notification* permettant d'exécuter des commandes sur une ressource de calcul.

L'opération *put* du sac *Commande* a été redéfinie pour pouvoir allumer ou éteindre une ressource de calcul identifiée par son *id*. Ce *put* prend en paramètre un tuple complètement instancié dont le motif est *(id, cmd)*. Il teste d'abord la valeur de la variable *cmd* pour voir si elle est valide. Ensuite, lorsque la *cmd* est égale à *éteindre*, le *put* éteint la ressource de calcul en utilisant la commande appropriée selon son système d'exploitation, qui est lu dans le sac *Configuration*. Par exemple, le Listing 6.2 présente la commande qui est utilisée par l'opération *put* pour éteindre, localement, une ressource de calcul dont le système d'exploitation est Ubuntu.

```
shutdown now
```

Listing 6.2 – Exemple de commande pour éteindre une ressource de calcul

Lorsque la valeur de *cmd* est égale à *allumer*, le *put* lit d'abord le mode de démarrage et l'adresse MAC de la ressource de calcul. Ensuite, il allume la ressource selon son mode de démarrage. Enfin, le *put* effectue un *Ping* sur la ressource de calcul et insère un tuple dans *Notification* pour spécifier le fait qu'elle est allumée ou pas. Si le mode de démarrage est égal à *priseIntelligente*, le *put* envoie la commande à la

prise qui est associée à la ressource de calcul. Si le mode démarrage est égal à *WoL* (Wake on LAN), le *put* utilise la commande *etherwake*. Par exemple, le Listing 6.3 montre la commande qui permet d'allumer une ressource de calcul dont l'adresse MAC est égale à `00:14:4F:F7:65:0C`, `eno1` est le nom de l'interface réseau de type ethernet de la ressource de calcul depuis laquelle le démarrage est effectué.

```
sudo etherwake -i eno1 00:14:4F:F7:65:0C
```

Listing 6.3 – Exemple de commande pour allumer une ressource de calcul

Objet de type *Logiciel* Il contient trois sacs *Commande*, *Configuration* et *Notification* permettant d'installer un logiciel sur une ressource de calcul. Le *put* du sac *Commande* a été redéfini. Il prend en paramètre le nom d'un logiciel et l'installe, localement, sur la ressource de calcul qui exécute l'objet. Pour ce faire, le *put* lit d'abord le système d'exploitation de la ressource de calcul, le nom d'utilisateur et le mot de passe associé dans le sac *Configuration*. Ensuite, il installe le logiciel, en utilisant la commande appropriée, et insère un tuple dans *Notification* pour notifier que le logiciel est installé. Par exemple, le Listing 6.4 montre la commande utilisée pour installer *Dia*[1] sur une ressource de calcul dont le système d'exploitation est Ubuntu. L'option `-y` spécifie que l'installation est à faire sans demande de confirmation.

```
sudo apt-get install -y dia
```

Listing 6.4 – Exemple de commande pour installer un logiciel

Objet de type *Objet* Il contient deux sacs *Commande* et *Configuration*. L'opération *put* du sac *Commande* a été redéfinie pour démarrer un objet, localement sur une ressource de calcul, l'arrêter ou le migrer vers une autre ressource de calcul. Le *put* prend en paramètre un tuple qui est complètement instancié sous la forme *(id_objet, config, id_rsc_dest, id_rsc_src, cmd)* et effectue la commande spécifiée.

Lorsque la commande est égale à *demarrer*, le *put* démarre localement l'objet sur la ressource de calcul de destination *id_rsc_dest*. Pour ce faire, le *put* récupère l'adresse IP de la ressource de calcul et utilise la commande *start_object* de LINC. Lorsque la commande est égale à *arreter*, le *put* arrête, localement, l'objet depuis la ressource de calcul sur laquelle il est exécuté, avec la commande *stop_object*. Enfin, lorsque la commande est égale à *migrer*, le *put* migre l'objet de la ressource source vers la ressource destination avec la commande *migrate_object* de LINC.

Objet de type *Regle* Il contient trois sacs, *CompileId*, *Commande* et *Trigger*, permettant d'activer et de désactiver des règles. Le sac *CompileId* contient des tuples, sous la forme *(id_regle, id_compile_regle)*, qui mappent l'*id* dans la description d'une règle à son identifiant de compilation généré par l'objet qui l'exécute. *Commande* contient des tuples qui sont sous la forme *(id_regle, commande)*. Le *put*

1. http ://dia-installer.de/index.html.fr

de ce sac a été redéfini. Il lit d'abord, dans le sac *CompileId*, l'identifiant de compilation associé à la règle. Ensuite, le *put* insère le tuple *(id_ compile_ regle, commande)* dans le sac *Trigger*. Cette insertion déclenche une règle LINC qui active ou désactive la règle spécifiée suivant le principe présenté dans la section 2.3 du chapitre 2.

6.1.2.2 Langage de description textuel

Le langage permettant de décrire un système et ses objectifs a été développé à l'aide de l'environnement Eclipse Xtext [16]. La grammaire du langage a d'abord été définie, sur la base du méta-modèle des systèmes considérés (cf. chapitre 2). Ensuite, un éditeur associé à ce langage été généré à l'aide de Xtext. Cet éditeur supporte l'auto-complétion, la vérification syntaxique et la coloration des mots clés.

6.1.2.3 Générateur de règles exécutant une fonction de transitions

La règle qui exécute la fonctions de transitions du contrôleur (fonction *step*), dans le cas d'une boucle de déploiement, gère de façon dynamique l'ordre d'exécution des commandes. Cet ordre dépend de la valeur des commandes et est défini à chaque fois que des commandes sont calculées. Cela est effectué comme présenté au chapitre 4. La *précondition* de la règle collecte, dans un premier temps, les données et appelle le *step* pour calculer les commandes à exécuter. Ensuite, la *précondition* appelle une fonction, appelée *genererTransaction*, qui définit l'ordre dans lequel les commandes calculées doivent être exécutées. Cette fonction génère une transaction qui est stockée dans une variable appelée *transact*. Enfin, la *performance* de la règle insère la transaction générée dans un sac particulier pour permettre son exécution.

Le générateur d'une telle règle est constitué de trois fonctions Python qui sont : *generer_ Precondition()*, *generer_ Transaction()* et *generer_ Performance()*. Ces fonctions génèrent, respectivement, la *précondition* de la règle, la transaction définissant l'ordre d'exécution des commandes qui sont calculées et la *performance* de la règle.

generer_Precondition() Cette fonction prend en entrée, les paramètres du *step* généré lors de la compilation du programme H/BZR du système. Elle génère :

- ***un rd*** d'un tuple dans un sac pour chaque entrée du *step* pour lire sa valeur ;
- ***un INLINE_ COMPUTE*** qui permet de stocker les entrées dans une variable appelée *entrees*. Cette variable est une chaîne de caractères qui est dans ce format $entree_1$_ *nom,*$entree_1$_ *val, ...,* $entree_n$_ *nom,*$entree_n$_ *val* ;
- ***un rd*** sur le sac *Step* de l'objet *HBZR* pour calculer les commandes à exécuter. Le tuple lu est le suivant *(entrees, etatCourant, commandes)*. Ce *rd* retourne l'état courant de l'automate et les commandes calculées dans les variables *etatCourant* et *commandes*. La variable *Commande* est une chaîne de caractères de ce format cmd_1_ *nom,*cmd_1_ *val, ...,*cmd_n_ *nom,*cmd_n_ *val* ;
- ***un COMPUTE*** qui appelle la fonction *genererTransaction()* , avec les variables *entree* et *commandes*, pour générer la transaction qui gère l'ordre d'exécution des commandes et la stocker dans une variable appelée *transact* ;

genererTransaction() Cette fonction formate dans un premier temps les variables *entrees* et *commandes* qu'elle reçoit en paramètre. Ensuite, elle analyse les commandes et définit l'ordre d'exécution comme décrit au chapitre 4. Enfin, cette fonction génère et retourne une transaction constituée des opérations suivantes :

- ***un rd***, pour chaque entrée pour vérifier si sa valeur est toujours valide ;
- ***un rd*** sur le sac *StepAccess* de l'objet *HBRZ*, pour verrouiller l'accès au *step* et empêcher qu'il soit exécuté par plusieurs instances de la même transaction ;
- ***un put*** d'un tuple dans un sac pour chaque commande, pour l'exécuter ;
- ***un put*** du tuple *(entrees, commandes,"")* dans le sac *Step* de l'objet *HBZR*, pour exécuter le *step* et changer l'état de l'automate modélisant le système.

generer_Performance() Cette fonction génère un *put* du tuple *(transact)* dans le sac *AddRules* de l'objet qui exécute la règle (*ego*). Le tuple *transact* correspond à la transaction générée pour l'ordre d'exécution des commandes. Le sac *AddRules*, comme détaillée dans [79], compile la transaction et l'exécute, de façon dynamique.

6.1.3 Implémentation de boucles applicatives

L'implémentation effectuée est relative à la couche d'abstraction et aux générateurs de règles exécutant une fonction de transitions : règles *template* et règles instances.

6.1.3.1 Générateur de couches d'abstraction

Ce générateur est une fonction qui prend en entrée le fichier de description du système. Il instancie pour chaque technologie de communication utilisée par les capteurs et/ou les actionneurs décrits, un objet de l'environnement d'abstraction PUTUTU de type spécifique. Le type de l'objet correspond au nom de la technologie de communication. Par exemple, pour la technologie de communication *EnOcean*, l'objet PUTUTU instancié est de type *EnOcean*. Cet objet, encapsulant la technologie *EnOcean*, permet de communiquer avec les capteurs et/ou actionneurs *EnOcean*.

6.1.3.2 Générateur de règles *templates*

Ce générateur est constitué de deux fonctions pour générer la *précondition* et la *performance* d'une *règle template* qui exécutent une fonctions de transitions. Il prend en entrée, le code C généré par la compilation du programme H/BZR considéré et génère la règle selon le principe décrit à la section 6.1.2.3. Les différences sont :

- **la règle *template*** lit et insère des tuples, contenant le mot *id*, sur des mémoires associatives, *template* (*memoireAssociative*) d'objets appelés *Obj* ;
- **la règle *template*** ne gère pas l'ordre d'exécution des commandes. La raison est que les commandes sont effectuées sur des actionneurs de bâtiments intelligents et il n'est pas nécessaire de gérer l'ordre. Par exemple, allumer une lampe, éteindre un chauffage et ouvrir une fenêtre peuvent être effectuées

dans n'importe quel ordre. Par conséquent, cette règle n'appelle pas, dans sa *précondition*, une fonction qui génère une transaction pour gérer l'ordre.

6.1.3.3 Générateur de règles instances

Ce générateur est également constitué de deux fonctions. Il prend en entrée, une règle template et le fichier de description des capteurs et actionneurs du système. Ce générateur remplace dans la règle *template*, *id*, *memoireAssociative* et *Obj*, respectivement, par les *id*, les noms de mémoires associatives et ceux d'objets spécifiques. Le remplacement est effectué en utilisant les informations sur les capteurs et actionneurs décrits (c.-à-d. *id*, emplacement et technologie de communication).

6.2 Études de cas

Cette section présente deux études de cas pour la validation de SICODAF. Chaque étude de cas est d'abord décrite. Ensuite une boucle est mise en œuvre pour sa conception ou son déploiement et le comportement de la boucle est présenté.

6.2.1 Déploiement d'un système de traitement de données

Ce système est une extension de l'exemple présenté au chapitre 4. Il est constitué d'une application de quatre fonctionnalités : *acquisition de données*, de cinq capteurs de température depuis un site web, *analyse de données*, *diffusion des résultats* et *maintien de la confidentialité*. Les fonctionnalités *acquisition de données* et *analyse de données* sont fournies par une tâche $T1$ qui utilise le logiciel Octave. La fonctionnalité *affichage des résultats* est offerte par deux tâches $T2$ et $T3$ qui utilisent respectivement un écran et une imprimante. La fonctionnalité *maintien de la confidentialité* est offerte par une tâche $T4$ qui efface le contenu de tous les écrans pendant les heures de visite. Lorsqu'elles sont actives au même instant, les tâches $T2$ et $T4$ sont en conflit sur l'écran (afficher et effacer au même instant). Ce conflit doit être évité, en empêchant que les tâches, $T2$ et $T4$, soient actives au même instant.

La plateforme d'exécution, associée à l'application de traitement de données, est composée de trois ressources de calcul ($D1$, $D2$, $D3$) et deux imprimantes ($P1$, $P2$), connectées via un réseau local. Les ressources de calcul $D1$ et D_2 peuvent être démarrées à l'aide du Wake on LAN (WoL) et D_2 a un écran E_1. $D3$ ne supporte pas le WoL et est associée à une prise intelligente qui permet de la démarrer.

6.2.1.1 Description des entités du système

Le tableau 6.1 décrit les entités de l'application de traitement de données. La tâche $T1$ possède deux versions ($T1.V1$ et $T1.V2$) qui collectent les données et les analysent. La version $T1.V1$ offre une qualité de service égale à `Elevee` et est mise en œuvre à l'aide d'une configuration d'objet ($O1.conf1$) et d'une règle ($R1$ qui communique avec $O1$). La version $T1.V2$ offre une qualité de service égale à `Moyenne` et est aussi mise en œuvre à l'aide d'une configuration d'objet ($O1.conf2$)

Tâches	Versions (qualité de service)	Transitions	Configurations d'objets	Règles	Objets	Ressources d'entrée/sortie
$T1$	$T1.V1$ (Elevee) $T1.V2$ (Moyenne)	$T1.V1 \leftrightarrow T1.V2$	$O1.conf1$ $O1.conf2$	$R1$ $R2$	$O1$	Octave
$T2$	$T2.V1$ (Elevee) $T2.V2$ (Moyenne) $T2.V3$ (Moyenne)	$T2.V1 \leftrightarrow T2.V2$ $T2.V2 \leftrightarrow T2.V3$	$O2.conf1$ $O2.conf2$ $O2.conf3$	$R3$	$O2$	écran
$T3$	$T3.V1$ (Elevee)	-	$O3.conf1$	$R4$	$O3$	imprimante
$T4$	$T4.V1$ (Elevee)	-	$O4.conf1$	$R5$	$O4$	écran

TABLE 6.1 – Application de traitement de données

Ressources de calcul	Système d'exploitation	Mode de démarrage	Ressources d'entrée/sortie (mode d'accès)	Type de (Ressources d'entrée/sortie)
$D1$	Ubuntu	wake on LAN	$P1$ (réseau), $P2$ (réseau)	imprimante
$D2$	Windows	wake on LAN	$E1$ (local), $P1$ (réseau), $P2$ (réseau)	écran, imprimante
$D3$	Ubuntu	prise intelligente	$P1$ (réseau), $P2$ (réseau)	imprimante

TABLE 6.2 – Plateforme d'exécution de l'application de traitement de données

et d'une règle ($R2$). L'objet $O1$ requiert le logiciel Octave. La tâche $T2$ possède trois versions ($T_2.V1$, $T2.V2$ et $T2.V3$) qui utilisent un écran et affichent les résultats avec une taille de police égale à `large`, `normale` et `petite`, respectivement. Pour le confort visuel, la tâche $T2$ possède les transitions suivantes : $T2.V1 \leftrightarrow T2.V2$ et $T2.V2 \leftrightarrow T2.V3$. La tâche $T3$ possède une version qui utilise une imprimante pour diffuser les résultats. La tâche $T4$ possède aussi une version. Cette version utilise un écran et efface son contenu pour maintenir la confidentialité. Les charges des configurations d'objets et celles des règles sont négligées dans ce système. La raison est que la quantité de données collectées (de cinq capteurs) n'est pas importante.

Le Tableau 6.2 décrit les entités de la plateforme d'exécution. Les ressources de calcul D_1, D_2, D_3 possèdent respectivement, Ubuntu, Windows et Fedora comme système d'exploitation. Pour le mode de démarrage, D_1 et D_2 supportent le Wake on LAN et D_3 est associée à une prise intelligente. Enfin, D_2 possède un écran, E_1, et toutes les ressources de calcul ont accès aux imprimantes P_1 et P_2 via le réseau. Chaque ressource de calcul possède des propriétés (p. ex., adresse MAC, adresse IP, nom d'utilisateur, mot de passe) qui ne sont pas présentées dans le Tableau 6.2.

6.2.1.2 Description des objectifs du système

Les objectifs devant être réalisés par le système de traitement de données sont :
— **objectif1** : les fonctionnalités *acquisition de données*, *analyse de données* et *affichage des résultats* doivent être fournies, en continu, de 6 h à 14 h ;
— **objectif2** : la fonctionnalité *maintien de la confidentialité* doit être fournie durant les heures de visite si l'écran $E1$ de la ressource $D2$ est disponible.

6.2.1.3 Mise en œuvre d'une boucle de déploiement

Le contrôleur est d'abord conçu à l'aide de la synthèse de contrôleurs discrets. Ensuite, une règle (exécutant la fonction de transitions) et la couche d'abstraction

sont générées. Enfin, le comportement de la boucle est montré à l'aide de chronogrammes.

Conception du contrôleur de la boucle Un modèle comportemental et un *contrat* son définis. Le modèle comportemental de ce système est composé de :

- quatre automates de tâches pour $T1$, $T2$, $T3$ et $T4$;
- quatre automates d'objets pour $O1$, $O2$, $O3$ et $O4$;
- cinq automates de règles pour $R1$, $R3$, $R4$ et $R5$;
- deux automates de ressources d'entrée/sortie matérielles pour $P1$ et $P2$;
- trois automates d'hôtes pour $D1$, $D2$ et $D3$. Chacun de ces automates d'hôtes est composé avec un automate de ressource d'entrée/sortie logicielle modélisant le logiciel *Octave*. L'automate d'hôte associé à $D2$ est composé à un automate de ressource d'entrée/sortie matérielle qui modélise l'écran $E1$.

Les automates de $T1$, $O1$, $R1$, $D1$ et l'automate modélisant le logiciel Octave correspondent à ceux qui ont été présentés à la section 4.1.3 du chapitre 4. La Figure 6.1 présente l'automate de la ressource d'entrée/sortie matérielle modélisant l'imprimante $P1$. Cet automate modélise le fait que l'imprimante est $P1$ est initialement `Inutilise`. Il peut devenir `Indisponible` ou `Utilise` en fonction des entrées de l'automate : `panne`, `dispo`, `c1`, `c3` où `c1` et `c3` sont des points de contrôlabilité. Les autres automates du modèle comportemental du système sont conçus sur le même principe, décrit dans la section 4.1.2 du chapitre 4, et ne sont pas présentés.

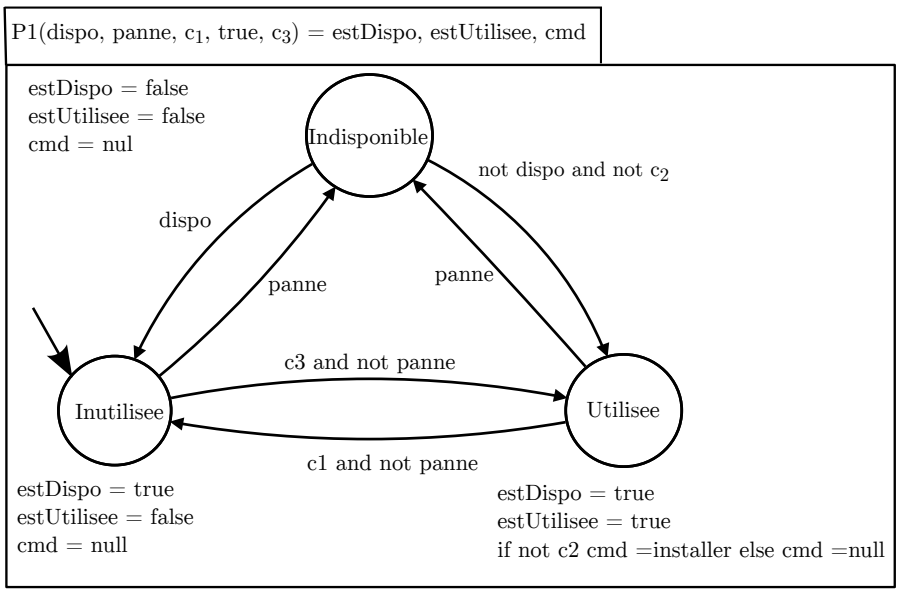

FIGURE 6.1 – Automate de l'imprimante $P1$

La Figure 6.2 présente le nœud H/BZR qui contient le *contrat* défini pour le système de traitement de données. Les entrées de ce nœud correspondent aux données collectées. Elles sont relatives au temps, à l'heure de visite, aux pannes, disponibilités et requêtes de maintenance relatives aux ressources de la plateforme d'exécution.

Les sorties de ce nœud quant à elles correspondent aux commandes à exécuter sur les entités du système (les objets, les règles, les ressources de calcul et d'entrée/sortie).

Le *contrat* de ce nœud définit deux hypothèses, des propriétés relatives aux entités du système et des variables contrôlables. Les hypothèses sont : (i) l'état où les ressources de calcul $D1$, $D2$ et $D3$ sont toutes les trois indisponibles ne peut pas être atteint, (ii) l'état où les imprimantes $P1$ et $P2$ sont toutes les deux indisponibles ne peut pas être atteint. Ces hypothèses garantissent que le système peut fonctionner. Elles stipulent qu'au moins une ressource de calcul et une imprimante sont disponibles. Cela permet de fournir les fonctionnalités lorsqu'elles doivent être fournies. Les variables contrôlables correspondent aux points de contrôlabilité des automates du modèle comportemental. Les propriétés sont quant à elles relatives aux tâches, aux objets, aux règles et aux ressources d'entrée/sortie matérielles (l'écran $E1$).

Propriétés relatives aux tâches Ces propriétés sont les suivantes :

— lorsque l'entrée `temps` est comprise entre 6 et 14, $T1$ doit être active. La raison est que les fonctionnalités *acquisition de données* et *analyse de données* doivent être fournies de 6 h à 14 h et $T1$ est l'unique tâche qui les offre ;

— lorsque l'entrée `temps` est comprise entre 6 et 14, l'une des tâches $T2$ ou $T3$ doit être active. La raison est que la fonctionnalité *affichage des résultats* doit être fournie, de 6h à 14h, et elle est offerte par chacune de ces deux tâches ;

— lorsque l'entrée `heureVisite` est égale à vraie et l'écran $E1$ est disponible, de même que la ressource de calcul $D2$, la tâche $T4$ doit être active. La raison est que la fonctionnalité *maintien de la confidentialité* doit être fournie ;

— les tâches $T2$ et T4 ne doivent pas être actives au même instant pour éviter le conflit sur l'écran (afficher résultats sur l'écran et effacer son contenu) ;

— lorsqu'une tâche est active, les objets de l'une de ses versions doivent être démarrés et les règles associées, à cette version, doivent être activées.

Propriétés relatives aux objets Les propriétés relatives aux objets sont :

— lorsque l'objet $O1$ est démarré, sa ressource de calcul de destination doit être allumée et le logiciel Octave doit y être disponible ($O1$ requiert ce logiciel) ;

— lorsque l'objet $O2$, de même que $O4$, est démarré, sa ressource de calcul de destination doit être allumée et doit avoir un écran. Dans ce cas, le nombre d'utilisateurs de l'écran augmente de un. Le nombre d'utilisateurs de l'écran est calculé car il ne doit pas être utilisé par plusieurs objets à la fois ;

— lorsque l'objet $O3$ est démarré, sa ressource de calcul de destination doit être allumée et l'une des deux imprimantes ($P1$, $P2$) doit être disponible.

Propriétés relatives aux règles et ressources d'entrée/sortie Les propriétés relatives aux règles spécifient que lorsqu'une des règles est active, l'objet qui l'exécute doit être démarré. Une seule propriété est définie pour les ressources d'entrée/sortie et elle est relative à l'écran $E1$. Cette propriétés spécifie que le nombre d'utilisateurs de $E1$ doit être inférieur ou égal à un (son mode d'utilisation est *1E*).

```
TraitementDonnees(temps, heureVisite, dispo_D1, panne_D1,rmaint_D1, dispo_D2, panne_D2,rmaint_D2,
dispo_D3, panne_D3,rmaint_D3, dispo_P1, panne_P1, dispo_P2, panne_P2, dispo_E1, panne_E1) =
(cmdD1, cmd_D2, cmd_D3, cmd_O1, rsc_dest_O1, rsc_source_O1, config_O1, cmd_O2, rsc_dest_O2,
rsc_source_O2, config_O2,cmd_O3, rsc_dest_O3, rsc_source_O3, config_O3,cmd_O4, rsc_dest_O4,
rsc_source_O4, config_O4, cmd_R1, objet_R1, cmd_R2, objet_R2, cmd_R3, objet_R3,cmd_R4, objet_R4
cmd_R5, objet_R5, cmd_Octave_D1,cmd_Octave_D2,cmd_Octave_D3)
contract
 assume
   not (D1.estDispo = false and D2.estDispo = false and D3.estDispo = false) and
   not (P1.estDispo = false and P2.estDispo = false)
 enforce
 #propriétés relatives aux tâches
   temps in [6, 14] => T1.active and
   temps in [6, 14] => (T2.active or T3.active) and
   (heureVisite and E1.estDispo and D2.estDispo) => T4.active and
   not (T2.active and T4.active) and
   Ti.active => (Ti.configurationObjets.demarre and Ti.regles.active)

 # propriétés relatives aux objets
  O1.demarre => (O1.rsc_dest.on and O1.rsc_dest.Octave.estDispo) and
  O2.demarre => (O2.rsc_dest.on and O2.rsc_dest.ecran.estDispo and O2.dest.ecran.nUtili = O2.dest.ecran.nUtiliPrec+1) and
  O3.demaree => (O3.rsc_dest.on and (P1.estDispo or P2.estDispo)) and
  O4.demarre => (O4.rsc_dest.on and O4.rsc_dest.ecran.estDispo and O4.dest.ecran.nUtili = O4.dest.ecran.nUtiliPrec+1)and

 # propriétés relatives aux règles
  Ri.active => Ri.objet.demarre and

 # propriétés relatives aux ressources d'enrée/sortie
  E1.nUtili <= 1

 with (c1_T1, c2_T1, c3_T1, c1_O1, c2_O1, c3_O1,crsc_O1,
         c1_R1, c2_R2, cobj_R1,c1_R1, c2_R2, cobj_R1,
         c1_ocatve_D1, c2_octave_D1, c3_octave_D1,
         c1_ocatve_D2, c2_octave_D2, c3_octave_D1,
         c1_D1, c2_D1, c3_D1, c1_D2, c2_D2, c3_D2, ...)
```

FIGURE 6.2 – *Contrat* du système de traitements de données

Règle exécutant le contrôleur de la boucle Le modèle comportemental et le *contrat* du système de traitement de données sont compilés et la fonction *step* générée est invoqué dans une règle LINC. Cette règle est présentée au Listing 6.5.

La règle collecte d'abord les entrées et les stocke dans la variable *entrees*. Ensuite, elle invoque la fonction *step* pour calculer les commandes et récupérer l'état courant de l'automate du système (ligne 19). Ensuite, la règle invoque la fonction *genererTransaction* et stocke la transaction qui permet d'exécuter les commandes dans la variable *transact* (ligne 20). Enfin, la règle exécute la transaction générée, grâce à son insertion dans le sac *AddRules* de l'objet qui l'exécute. La règle est déclenchée à chaque fois qu'une nouvelle valeur est produite pour une entrée.

```
1  ["Systeme","Temps"].rd("temps",temp_val)&
   ["Systeme","HeureVisite"].rd("heureVisite", heureVisite_val)&
3  ["Maintenance","RequeteMaintenance"].rd("D1",rmaint_D1)&
   ["Detection","Etat"].rd("D2",panne_D2,dispo_D2)&
5  ["Maintenance","RequeteMaintenance"].rd("D2",rmaint_D2)&
   ["Detection","Etat"].rd("D3",panne_D1,dispo_D3)&
7  ["Maintenance","RequeteMaintenance"].rd("D3",rmaint_D3)&
   ["Detection","Etat"].rd("P1",panne_P1,dispo_P1)&
9  ["Detection","Etat"].rd("P2",panne_P2,dispo_P2)&
   ["Detection","Etat"].rd("E1",panne_E1,dispo_E1)&
11 INLINE_COMPUTE: entrees = "(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,
   %s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)"
13 %("temps",temp_val,"heureVisite", heureVisite_val,
   "panne_D1",panne_D1,"dispo_D1",dispo_D1,"rmaint_D1",rmaint_D1,
15 "panne_D2",panne_D2,"dispo_D2",dispo_D2,"rmaint_D2",rmaint_D2,
   "panne_D3",panne_D3,"dispo_D3",dispo_D3,"rmaint_D3",rmaint_D3,"panne_P1",
17 panne_P1,"dispo_P1",dispo_P1,"panne_P2",panne_P2,"dispo_P2",dispo_P2,
   "panne_E1",panne_E1,"dispo_E1",dispo_E1)&
19 ["HBZR","Step"].rd(entrees,etatCourant,commandes)&
   COMPUTE: transact = genererTransaction(entrees,etatCourant,commandes)
21 ::
   {
23  [ego, "AddRules"].put(transact);
   }.
```

Listing 6.5 – Règle du système de traitement de données

Couche d'abstraction de la boucle La couche d'abstraction est constituée de :

- quatre instances d'objets LINC de type *Materiel* : une instance pour chaque ressource de calcul ($Materiel\_D1, Materiel\_D2, Materiel\_D3$) pour l'éteindre et une instance ($Materiel\_General$) qui permet de les allumer ;
- trois instances d'objet LINC de type *Logiciel* pour chaque ressource de calcul ;
- trois instances d'objets LINC de type *Objet* pour chaque ressource de calcul ;
- trois instances d'objets LINC de type *Regle* pour chaque ressource de calcul.

6.2.1.4 Comportement de la boucle

La Figure 6.3 présente une trace d'exécution du contrôleur de la boucle du système. Cette trace d'exécution est obtenu à l'aide l'outil Sim2chro de l'équipe Verimag [2]. Elle montre les états des tâches et des ressources de calcul (variables en rouge) du système en fonction des données qui sont collectées (variables en bleu).

De 1 h à 6 h, ce n'est pas encore l'heure des visites, toutes les ressources de calcul sont disponibles. Comme aucune fonctionnalité ne doit être fournie, toutes les tâches sont inactives et toutes les ressources de calcul sont éteintes. À 12h, ce n'est pas encore l'heure des visites et les ressources de calcul sont toujours disponibles. Pour réaliser l'objectif de la boucle qui consiste à fournir les fonctionnalités *acquisition de données*, *analyse de données* et *affichage des résultats* de 6 h à 14h, le contrôleur active les tâches $T1$ et $T3$ et démarre la ressource de calcul $D1$. La

2. http ://www-verimag.imag.fr/ ?lang=fr

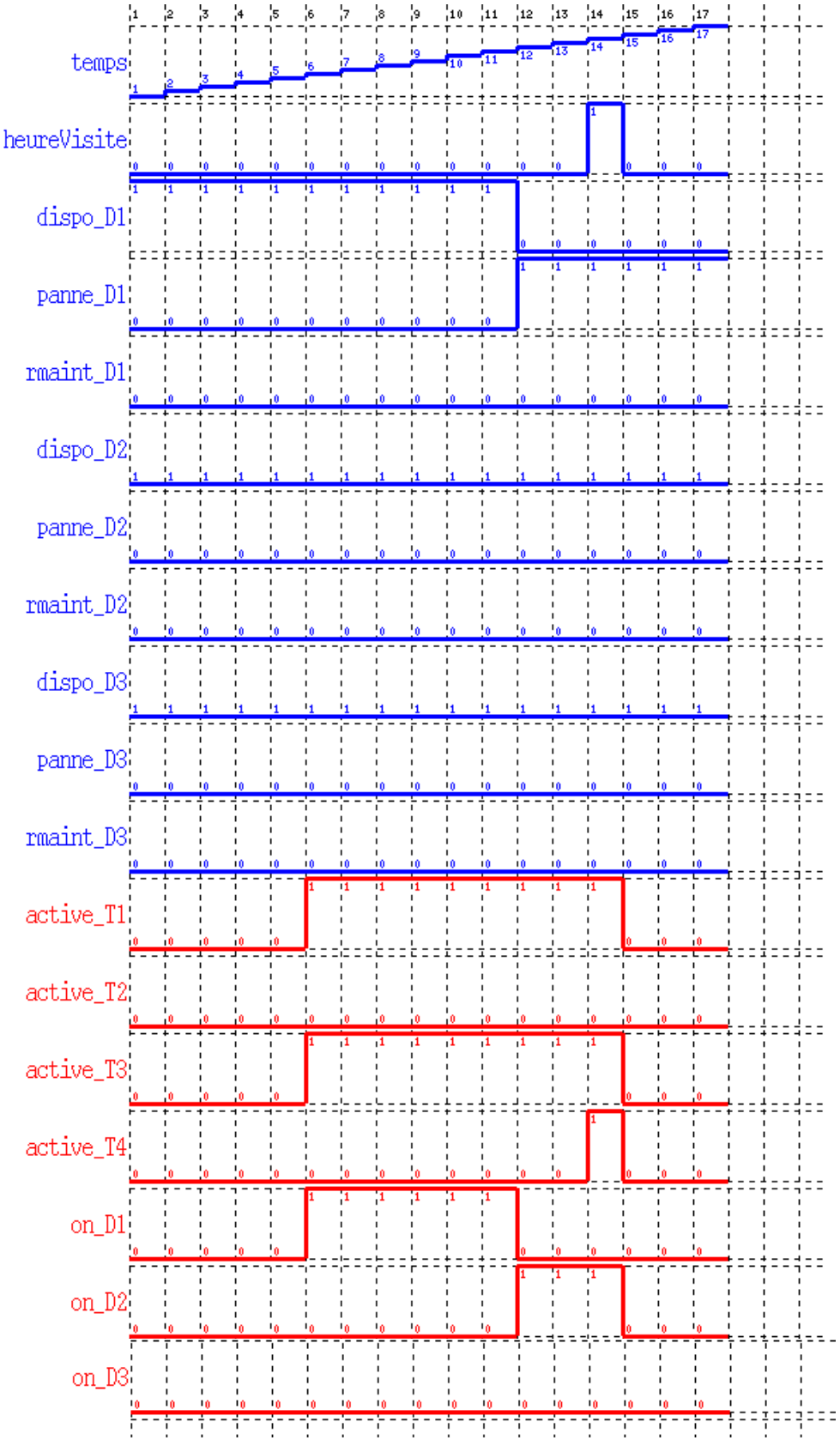

FIGURE 6.3 – Chronogramme du système de traitements de données

tâche $T1$ fournit les deux premières fonctionnalités et $T3$ fournit la fonctionnalité *affichage des résultats*.

Le contrôleur aurait pu choisir d'activer $T2$ à la place de $T3$ mais dans ce cas, il faudrait allumer la ressource de calcul $D2$ car $T2$ a besoin d'un écran et $D1$ n'en possède pas. À 12 h, la ressource de calcul $D1$ tombe en panne. Dans cas, pour continuer à fournir les trois fonctionnalités, le contrôleur allume la ressource de calcul $D2$ et y déploie les objets et règles des tâches $T1$ et $T3$. À 14 h, l'entrée relative aux heures de visite est égale à *vrai*. Pour fournir la fonctionnalité *maintien de la confidentialité*, le contrôleur active la tâche $T4$ qui utilise un écran et efface son contenu. Le contrôleur n'allume pas une nouvelle ressource de calcul car $D2$ dispose d'un écran et est déjà allumée. À 15 h, l'entrée relative aux heures de visite devient égale à *faux* et le contrôleur désactive la tâche $T4$ (la fonctionnalité *maintien de la confidentialité* ne doit plus être fournie). Cependant, le contrôleur n'éteint pas $D2$ car les tâches $T1$ et $T3$ sont toujours actives et leurs objets et règles y sont déployés. Enfin, à partir de 15 h, aucune fonctionnalité ne doit être fournie, le contrôleur désactive les tâches $T1$ et $T2$ et éteint la ressource de calcul $D2$.

La Figure 6.3 montre que le contrôleur de la boucle calcule des commandes correctes et cohérentes permettant d'adapter le système de traitements de données.

6.2.2 Conception d'une pièce de bureau intelligente

La pièce de bureau considérée est équipée de plusieurs capteurs et actionneurs. Les actionneurs doivent être contrôlés, de façon automatique, dans le but d'assurer le confort des occupants et de minimiser la consommation d'énergie de la pièce.

6.2.2.1 Description des capteurs et des actionneurs

La pièce *Piece*1, comme présentée au Listing 6.6, est équipée d'une fenêtre, d'un volet, d'une porte, d'une lampe, d'un climatiseur réversible, d'une ventilation mécanique et de trois capteurs intérieurs (présence, température et CO_2). Des capteurs sont également installés à l'extérieur de la pièce pour collecter des données relatives aux conditions extérieures (température, bruit, CO_2, luminosité et pollen) et un capteur de bruit est installé dans le couloir. Tous les capteurs utilisent la technologie de communication *TelosB* et les actionneurs sont de la technologie *EnOcean*. Les informations sur les réunions qui ont lieu dans la pièce sont fournies par un agenda. Elles permettent de savoir si une réunion est en cours, si elle est confidentielle et si une réunion va avoir lieu dans moins de trente minutes, après une réunion précédente.

```
# type:id:technologie:emplacement

fenetre:fenetre1:EnOcean:interieur

volet:volet1:EnOCean:interieur

porte:porte1:EnOcean:interieur

lamp:lamp1:EnOcean:interieur

climatiseurReversible:climRev1:EnOcean:interieur

ventilationMecanique:ventilMec1:EnOcean:interieur

presence:presenceInt1:TelosB:interieur

temperature:tempInt1:TelosB:interieur

CO2:CO2Int1:TelosB:interieur

temperature:tempExt1:TelosB:exterieur

bruit:bruitExt1:TelosB:exterieur

CO2:CO2Ext1:TelosB:exterieur

luminosite:lumExt1:TelosB:exterieur

pollen:pollenExt1:TelosB:exterieur

bruit:bruitCoul1:TelosB:exterieur
```

Listing 6.6 – Capteurs et actionneurs de la pièce *Piece*1

6.2.2.2 Description des objectifs

Les objectifs devant être réalisés dans la pièce de bureau sont les suivants :

- **Pour le confort**, lorsqu'une présence est détectée, la luminosité doit être entre 500 et 600 lux et le niveau de bruit doit être inférieur à 80 dB. Lorsqu'une présence est détectée et que la température est inférieure à 17 °C (resp. supérieure à 27 °C), la pièce doit être réchauffée (resp. refroidie) ;
- **Pour la qualité de l'air**, lorsqu'une présence est détectée et que le CO_2 dépasse 800 ppm, la pièce doit être ventilée. De plus, la pièce ne doit pas être polluée par le pollen ou le CO_2 extérieur. Enfin, la pièce doit être ventilée rapidement entre deux réunions qui sont séparées par moins de trente minutes ;
- **Pour la confidentialité**, la pièce doit être complètement fermée (la porte, la fenêtre et le volet doivent être fermés) durant une réunion confidentielle ;
- **Pour l'économie d'énergie**, l'éclairage, la ventilation, le chauffage et le refroidissement naturels sont préférés à l'éclairage, la ventilation, le chauffage et le refroidissement artificiels.

```
1  ["Piece1","Etat"].rd("volet1","ferme")&
   ["TelosB","Sensors"].rd("presenceInt1","True")&
3  ["TelosB","Sensors"].rd("lumExt1",lumExt_val)&
   INLINE_ASSERT: lumExt_val >= "500" and lumExt_val <= "600"&
5  ::
   {
7  ["TelosB","Sensors"].rd("presenceInt1","True")
   ["TelosB","Sensors"].rd("lumExt1",lumExt_val);
9  ["EnOcean","Actuators"].put("volet1","ouvrir");
   ["Piece1","Etat"].get("volet1","ferme");
11 ["Piece1","Etat"].put("volet1","ouvert");
   }.
```

Listing 6.7 – Exemple de règle de luminosité

6.2.2.3 Mise en œuvre d'une boucle applicative

La boucle de la pièce est d'abord mise en œuvre et un démonstrateur est présenté. Ensuite, des boucles sont conçues pour sept autres pièces, du même type mais avec plus d'actionneurs, et une comparaison des coûts de validation est effectuée.

Mise en œuvre de la boucle de la pièce Le contrôleur est, d'abord, mis en œuvre sous la forme de règles puis à l'aide de la synthèse de contrôleurs discrets et une comparaison est effectuée. Ensuite, la couche d'abstraction est mise en œuvre.

Contrôleur sous la forme de règles Pour réaliser les objectifs de la pièce, trente deux règles LINC ont d'abord été écrites. Chaque règle vérifie des conditions et envoie une commande à un actionneur. Les règles sont regroupées en cinq sous-ensembles : *Luminosité*, *Bruit*, *Qualité de l'air*, *Température* et *Confidentialité*.

Luminosité : ce sous-ensemble contient cinq règles qui envoient des commandes au volet et à la lampe pour maintenir la luminosité de la pièce entre 500 et 600 lux lorsqu'une présence est détectée. Ces règles sont : $RL1$, $RL2$, $RL3$, $RL4$ et $RL5$.

La règle $RL1$ est présentée au Listing 6.7. Cette règle vérifie d'abord si le volet est fermé et une présence est détectée. Ensuite, elle lit la valeur de la luminosité extérieure dans la variable *lumExt_val* et utilise l'opérateur *INLINE_ASSERT* de LINC pour vérifier si elle est comprise entre 500 et 600 lux. Si ce n'est pas le cas, l'exécution de la règle s'arrête. Sinon, la règle envoie la commande *ouvrir* au volet et met à jour son état logique. Cela est effectué de façon atomique, avec une transaction.

La règle $RL2$ allume la lampe lorsqu'une présence est détectée et que la luminosité extérieure n'est pas comprise entre 500 et 600. La règle $RL3$ ferme le volet lorsqu'il est ouvert, une présence est détectée et que la luminosité extérieure est supérieure à 600 lux. $RL4$ éteint la lampe lorsqu'elle est allumée et qu'une présence n'est plus détectée. $RL5$ éteint la lampe lorsqu'elle est allumée, une présence est détectée et la luminosité extérieure est entre 500 et 600 lux et le volet est ouvert.

```
["Piece1","Etat"].rd("fenetre1","ouverte")&
["TelosB","Sensors"].rd("presenceInt1","True")&
["TelosB","Sensors"].rd("bruitExt1",bruitExt_val)&
INLINE_ASSERT: bruitExt_val > "80" &
::
{
["Piece1","Etat"].rd("fenetre1","ouverte");
["TelosB","Sensors"].rd("presenceInt1","True");
["TelosB","Sensors"].rd("bruitExt1",bruitExt_val);
["EnOcean","Actuators"].put("fenetre1","fermer");
["Piece1","Etat"].get("fenetre1","ouverte");
["Piece1","Etat"].put("fenetre1","fermee");
}.
```

Listing 6.8 – Exemple de règle de bruit

Bruit : ce sous-ensemble contient deux règles $RB1$ et $RB2$. La règle $RB1$ ferme la fenêtre lorsqu'elle est ouverte, une présence est détectée dans la pièce et le bruit extérieur est supérieur à 80 dB. $RB2$, quant à elle, ferme la porte lorsqu'elle est ouverte, une présence est détectée et que le bruit du couloir est supérieur à 80 dB.

Le Listing 6.8 présente la règle $RB1$. Elle vérifie d'abord si la fenêtre est ouverte et une présence est détectée. Ensuite, elle lit le niveau du bruit extérieur et vérifie s'il est supérieur ou égal à 80 dB. Si c'est le cas, elle ferme la fenêtre, en effectuant un *put* sur le sac *Actuators* de l'objet *EnOcean*, et met à jour son état logique.

Qualité de l'air : ce sous-ensemble contient douze règles. Deux de ces règles empêchent le fait que la pièce soit polluée par le CO_2 extérieur et le pollen, en fermant la fenêtre. Les neuf autres règles envoient des commandes au volet, à la fenêtre et à la ventilation mécanique. Elles spécifient quand est ce que la fenêtre doit être ouverte, le volet doit être ouvert, la ventilation mécanique doit être démarrée et quand est ce qu'elle doit être arrêtée. Le nombre de règles est égal à neuf car il faut considérer tous les cas possibles. Par exemple, utiliser le volet et la fenêtre pour ventiler la pièce, lorsque cela est nécessaire, requiert de considérer trois cas :

— le volet et la fenêtre sont tous les deux fermés, dans ce cas il faut les ouvrir ;
— le volet est fermé et la fenêtre est ouverte, dans ce cas il faut ouvrir le volet ;
— le volet est ouvert et la fenêtre est fermée, dans ce cas il faut ouvrir la fenêtre.

Le Listing 6.9 présente la règle qui éteint la ventilation mécanique lorsque le CO_2 de la pièce est devenu inférieur à 600 ppm. Cette règle vérifie d'abord si la ventilation mécanique est allumée et lit la valeur du CO_2. Ensuite, elle vérifie que la valeur est inférieure à 600 ppm, éteint la ventilation et change son état logique.

Température : ce sous ensemble contient dix règles qui permettent de refroidir et de réchauffer la pièce, en envoyant des commandes au volet, à la fenêtre et au climatiseur réversible lorsque les conditions qu'elles vérifient sont vraies. Ces règles spécifient quand est ce que : la fenêtre (resp. le volet) doit être ouverte (resp. ouvert), le climatiseur doit être démarré (en mode chauffage ou refroidissement) et arrêté.

```
1  ["Piece1","Etat"].rd("ventilMec1","demarre")&
   ["TelosB","Sensors"].rd("CO2Int1",CO2Int_val)&
3  INLINE_ASSERT: CO2Int_val < "600" &
   ::
5  {
   ["Piece1","Etat"].rd("ventilMec1","demarre");
7  ["TelosB","Sensors"].rd("CO2Int1",CO2Int_val);
   ["EnOcean","Actuators"].put("ventilMec1","arreter");
9  ["Piece1","Etat"].get("ventilMec1","demarre");
   ["Piece1","Etat"].put("ventilMec1","arrete");
11 }.
```

Listing 6.9 – Exemple de règle de qualité de l'air

```
1  ["Piece1","Etat"].rd("volet1","ferme")&
   ["Piece1","Etat"].rd("fenetre1","fermee")&
3  ["TelosB","Sensors"].rd("presenceInt1","True")&
   ["TelosB","Sensors"].rd("tempInt1",tempInt_val)&
5  INLINE_ASSERT: tempInt_val > "27"&
   ["TelosB","Sensors"].rd("tempExt1",tempExt_val)&
7  INLINE_ASSERT: tempExt_val < tempInt_val
   ::
9  {
   ["TelosB","Sensors"].rd("presenceInt1","True")
11 ["TelosB","Sensors"].rd("tempInt1",tempInt_val);
   ["TelosB","Sensors"].rd("tempExt1",tempExt_val);
13 ["EnOcean","Actuators"].put("volet1","ouvrir");
   ["EnOcean","Actuators"].put("fenetre1","ouvrir");
15 ["Piece1","Etat"].get("volet1","ferme");
   ["Piece1","Etat"].put("volet1","ouvert");
17 ["Piece1","Etat"].get("fenetre1","fermee");
   ["Piece1","Etat"].put("fenetre1","ouverte");
19 }.
```

Listing 6.10 – Exemple de règle de température

Le Listing 6.10 présente la règle qui ouvre le volet et la fenêtre pour refroidir la pièce. Cette règle vérifie d'abord si le volet est fermé, la fenêtre est fermée et une présence est détectée dans la pièce. Ensuite, elle lit la valeur de la température intérieure et vérifie si elle est supérieure à 27 °C. Ensuite, la règle lit la valeur de la température extérieure et vérifie si elle est inférieure à la température intérieure. Si c'est le cas, la règle ouvre le volet et la fenêtre et met à jour leurs états logiques.

Confidentialité : ce sous-ensemble est constitué de trois règles qui ferment le volet, la fenêtre et la porte de la pièce durant une réunion confidentielle. Le listing 6.11 présente la règle qui ferme la porte. Cette règle vérifie si la porte est ouverte et une réunion confidentielle est en cours puis elle ferme la porte et change son état logique.

Les règles écrites pour réaliser les objectifs de la pièce *Piece*1 contiennent quarante deux conflits potentiels. Des exemples de tels confits sont :

— sur la fenêtre : ouvrir pour refroidir la pièce et fermer du fait du pollen ;
— sur le volet : ouvrir pour éclairer la pièce et fermer pour la confidentialité ;

```
["Piece1","Etat"].rd("porte1","ouverte")&
["Piece1","Agenda"].rd("True","True","")
::
{
["Piece1","Etat"].rd("porte1","ouverte");
["Piece1","Agenda"].rd("True","True","");
["EnOcean","Actuators"].put("porte","fermer");
["Piece1","Etat"].get("porte1","ouverte");
["Piece1","Etat"].put("porte1","fermee");
}.
```

Listing 6.11 – Exemple de règle de confidentialité

— sur la fenêtre : ouvrir pour ventiler la pièce et fermer du fait du bruit extérieur.

Pour garantir la fiabilité comportementale, les conflits ont été résolus, manuellement. Pour chaque conflit potentiel, une ou plusieurs opérations *rd* ont été ajoutées dans la *précondition* des règles impliquées, pour s'assurer qu'elles ne seront pas déclenchées au même instant, et de nouvelles règles ont été définies. Par exemple, la règle de luminosité $RL1$ a été modifiée comme suit : elle vérifie si une réunion confidentielle n'est pas en cours avant d'ouvrir le volet. Une nouvelle règle a été rajoutée pour allumer la lampe durant une réunion confidentielle lorsque la luminosité extérieure est comprise entre 500 et 600 lux. La raison est que $RB1$ ne sera pas déclenchée.

Au total, quarante et une conditions supplémentaires ont été ajoutées sur onze règles (34.37%) parmi trente deux et quatorze autres règles ont été ajoutées (43,75%).

Contrôleur basé sur la synthèse de contrôleurs discrets Un programme H/BZR, permettant de réaliser les objectifs de la pièce à l'aide de la synthèse de contrôleurs discrets, est d'abord défini. Ensuite, ce programme est compilé et le *step* généré est exécuté dans une règle *template*. Enfin, une instance de la règle *template* est générée à partir des informations sur les capteurs et actionneurs de la pièce.

Le programme H/BZR de la pièce est constitué d'un automate pour chaque type d'actionneur et d'un *contrat* qui réalise les objectifs. Chaque automate décrit les états du type d'actionneur, ses transitions d'états et ses effets sur les paramètres d'environnement qui sont considérés (luminosité, bruit, CO_2, température, pollen et air). L'automate modélisant le comportement de la lampe (resp. du volet) est celui qui est présenté à la Figure 4.11 (resp. la Figure 4.12) du chapitre 4.

La Figure 6.4 présente l'automate qui modélise la porte. Cet automate a deux états et deux transitions. Chaque état est associé à deux équations pour produire la commande de la porte et spécifier son effet sur le niveau de bruit d'une pièce. Par exemple, à l'état `Ouverte`, la porte introduit le bruit du couloir dans la pièce. Les effets de la porte sur les autres paramètres de l'environnement (air, CO_2, température, luminosité) ne sont pas pris en compte car les capteurs correspondants ne sont pas installés dans le couloir. Les transitions qui vont d'un état à un autre sont associées à `not c`. Cela permet d'ouvrir ou de fermer la porte seulement si nécessaire. Cela est rendu possible par le fait que le contrôleur généré est déterministe et il favorise

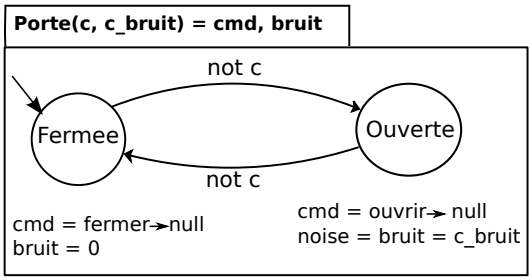

FIGURE 6.4 – Automate d'une porte

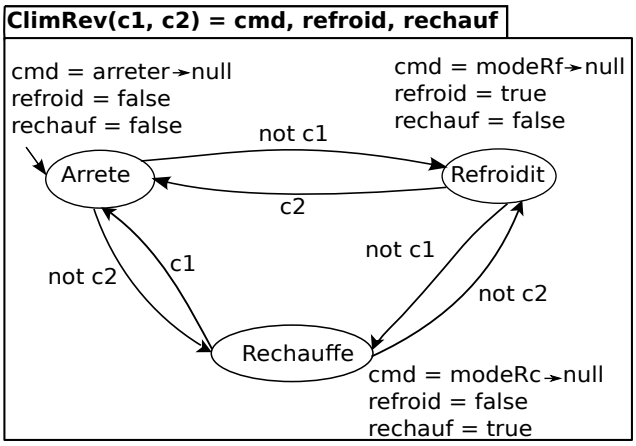

FIGURE 6.5 – Automate d'un climatiseur réversible

la valeur *true* à la valeur *false* pour une variable contrôlable booléenne.

La Figure 6.5 montre l'automate modélisant le climatiseur réversible (climRev). Cet automate a trois états et six transitions. Chaque état est associé à trois équations pour produire le commande du climRV et spécifier ses effets sur la pièce. Par exemple, à l'état `Arrete`, le climRV ne refroidit pas et ne chauffe pas la pièce. Cet automate possède deux entrées `c1` et `c2`. La raison est que, à chaque état, trois transitions peuvent être déclenchées (deux transitions qui quittent l'état et une qui permet de rester). Pour associer une expression booléenne différente à chacune des trois transitions d'un état, au moins deux variables sont nécessaires. Par exemple, lorsque l'état `Arrete` est activé, si l'entrée `c1` est égale à `false`, l'automate va dans à l'état `Refroidit`. Si `c2` est égale à `false`, il va dans l'état `Rechauffe`. Si `c1` et `c2` sont toutes les deux égales à *true*, l'automate reste dans l'état `Arrete`. Enfin, si `c1` et `c2` sont toutes les deux égales à `false`, la transition qui a été déclarée en premier est choisie. Le fait d'associer `not c1` et `not c2` (resp. `c1` et `c2`) aux transitions qui quittent (resp. arrivent à) l'état `Arrete` spécifie qu'il est préférable de maintenir le climRV à l'état `Arrete` pour minimiser la consommation d'énergie.

La Figure 6.6 présente l'automate modélisant la ventilation mécanique (VM). Cet automate a trois états. Chaque état est associé à trois équations pour produire la commande de la VM et spécifier son effet sur le CO_2 de la pièce. Par exemple, à

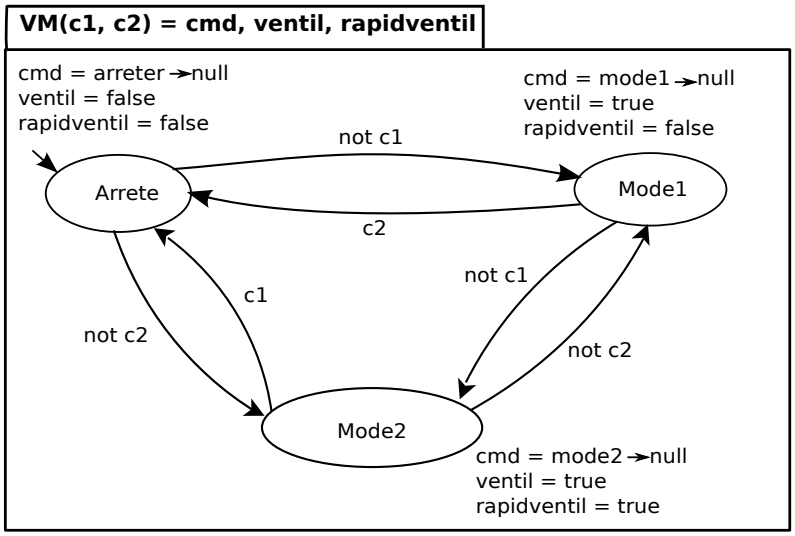

FIGURE 6.6 – Automate d'une ventilation mécanique

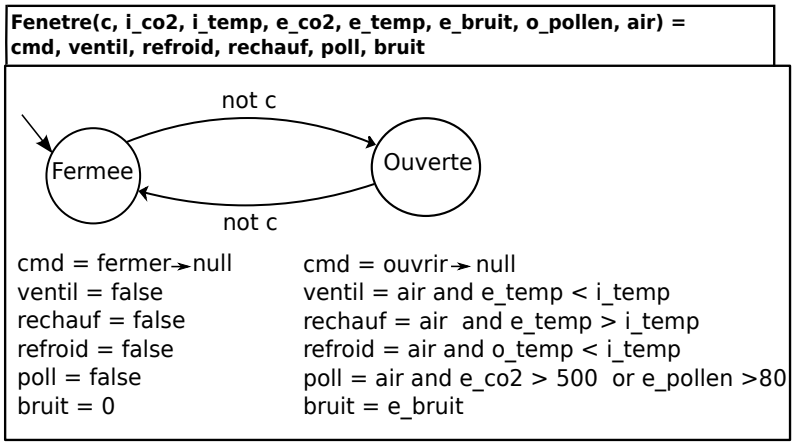

FIGURE 6.7 – Automate d'une fenêtre

l'état `Arrete`, la VM ne ventile pas la pièce. Au `Mode1`, elle ventile la pièce mais pas rapidement, comme au `Mode2`. Les transitions qui partent de `Arrete` sont associées à `not c1` et `not c2` pour spécifier qu'il est préférable de ne pas utiliser la VM.

La Figure 6.7 présente l'automate qui modélise une fenêtre. Cet automate a deux états. Chaque état est associé à cinq variables pour spécifier les effets de la fenêtre sur différents paramètres d'une pièce. A l'état `Fermee`, la fenêtre ne permet pas de ventiler, réchauffer ni refroidir la pièce. Elle ne pollue pas la pièce et n'y introduit pas le bruit extérieur. Les transitions qui conduisent à un état différent sont associés à `not c` pour spécifier que la fenêtre ne doit être ouverte ou fermée que si nécessaire.

La Figure 6.8 présente le nœud H/BZR qui contient le *contrat*. Ce *contrat* spécifie, de façon déclarative, les objectifs de la pièce et des variables contrôlables. Ces variables correspondent aux entrées des automates dont les valeurs ne sont pas

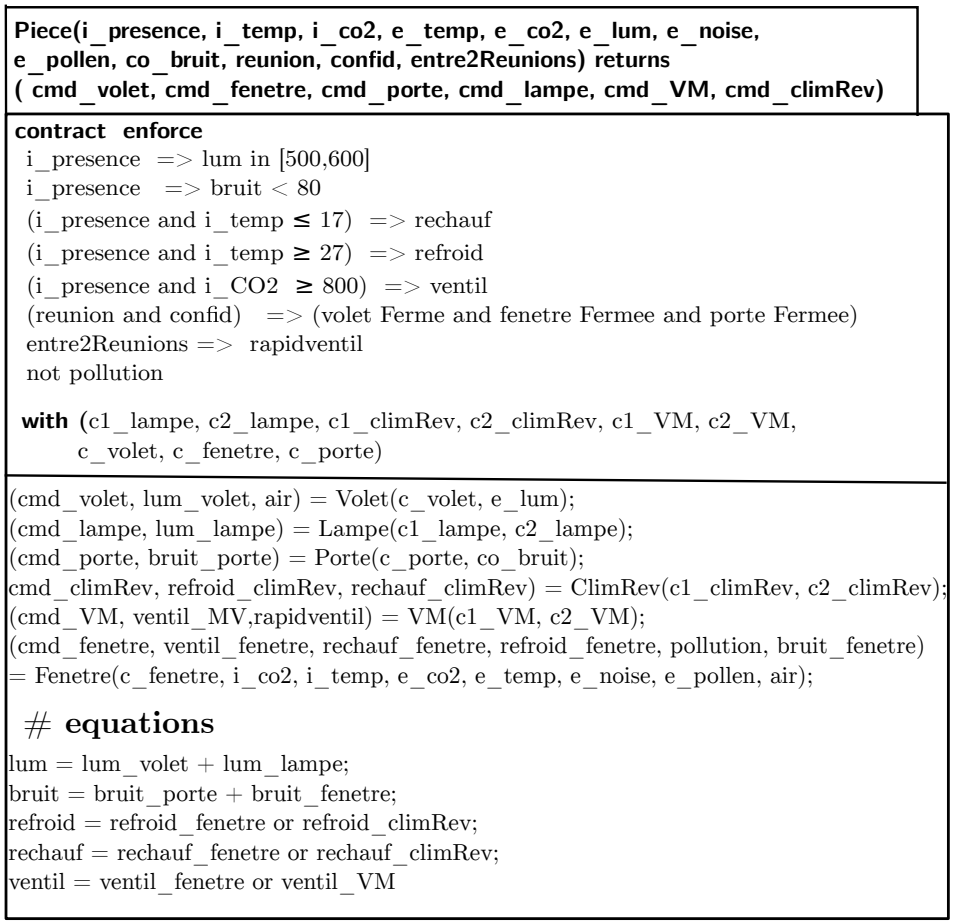

FIGURE 6.8 – Nœud H/BZR avec le *contrat* de la pièce *Piece*1

fournies par les capteurs ou l'agenda. L'objectif d'économie d'énergie (éclairage, chauffage, ventilation et refroidissement naturels sont préférés) est exprimé en déclarant les variables contrôlables du volet et de la fenêtre après celles de la lampe, du climRev et de la MV. L'ensemble des objectifs sera réalisé par la synthèse de contrôleurs discrets. Cela permet d'éviter les conflits et les violations d'objectifs.

Le Listing 6.12 montre la règle *template* qui a été générée pour exécuter le *step* qui résulte de la compilation du programme H/BZR de la pièce. La règle instance de la pièce n'est pas présentée. Elle est générée comme décrite à la section 6.1.3.3.

```
[Obj,"Sensors"].rd(id,i_pres_id_val)&
[Obj,"Sensors"].rd(id,i_temp_id_val)&
[Obj,"Sensors"].rd(id,i_co2_id_val)&
[Obj,"Sensors"].rd(id,e_temp_id_val)&
[Obj,"Sensors"].rd(id,e_co2_id_val)&
[Obj,"Sensors"].rd(id,e_lum_id_val)&
[Obj,"Sensors"].rd(id,e_bruit_id_val)&
[Obj,"Sensors"].rd(id,e_pollen_id_val)&
[Obj,"Sensors"].rd(id,co_bruit_id_val)&
["Piece","Agenda"].rd(reunion,confid,entre2Reunions)&
INLINE_COMPUTE: entrees= "(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)"
    %(i_pres_id_val,
i_temp_id_val,i_co2_id_val,e_temp_id_val,e_co2_id_val,
e_lum_id_val,e_bruit_id_val,e_pollen_id_val,
co_noise_id_val,reunion,confid,entre2Reunions)&
["HBZR","Step"].rd(entrees,etatCourant,commandes)&
INLINE_COMPUTE: cmd_volet,cmd_fenetre,cmd_porte,cmd_lampe,
cmd_VM,cmd_climRev = eval(commands)
::
{
#operations rd pour verifier les valeurs des donnees
[Obj,"Actuators"].put(id,cmd_volet);
[Obj,"Actuators"].put(id,cmd_fenetre);
[Obj,"Actuators"].put(id,cmd_porte);
[Obj,"Actuators"].put(id,cmd_lampe);
[Obj,"Actuators"].put(id,cmd_VM);
[Obj,"Actuators"].put(id,cmd_climRev);
["HBZR","Step"].put(entrees,etatCourant,"");
}.
```

Listing 6.12 – Règle *template* de la pièce *Piece*1

Comparaison des deux types de contrôleurs Lors de la mise en œuvre du contrôleur sous la forme de règles LINC, il a fallu considérer tous les cas possibles pour écrire un ensemble de règles complet. Il a aussi été nécessaire de modifier l'ensemble de règles pour éviter les conflits. Cela a été effectué en ajoutant des conditions supplémentaires sur plusieurs règles et en écrivant de nouvelles règles. Plusieurs itérations ont été faites pour s'assurer que tous les conflits ont été résolus.

Lors de la mise en œuvre à l'aide de la synthèse de contrôleurs discrets, il a fallu concevoir un automate pour chaque type d'actionneur et définir les objectifs à réaliser sous la forme d'un *contrat*. Cela a permis la génération d'un contrôleur. Il n'a pas été nécessaire de considérer tous les cas possibles et de résoudre des conflits. Cela est effectué, en mode hors ligne, par l'algorithme de synthèse de contrôleurs discrets. Par conséquent, le coût de la synthèse du contrôleur, même s'il est élevé, est un coût hors ligne. De plus, cela garantit l'absence de conflits et de violations.

Mise en œuvre de la couche d'abstraction La couche d'abstraction est constituée de deux objets PUTUTU : *TeloB* et *EnOcean*. Ces objets permettent respectivement de communiquer avec les capteurs et les actionneurs de la pièce.

Démonstrateur Pour illustrer la boucle de la pièce *Piece*1, un démonstrateur a été mis en œuvre. Le but est de réaliser les objectifs relatifs à la luminosité et à la

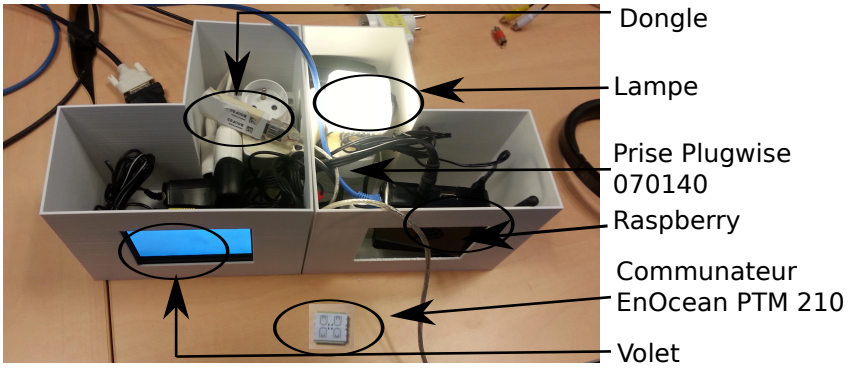

FIGURE 6.9 – Démonstrateur

confidentialité avec des capteurs et actionneurs concrets. Ces objectifs sont : *présence ⇒ luminosité dans [500 600] lux* et *réunion confidentielle⇒ pièce complètement fermée*. Le démonstrateur, comme représenté sur la Figure 6.9, est constitué :

- **d'une prise intelligente Plugwise** [106] : elle est utilisée pour allumer ou éteindre automatiquement la lampe. Cela est fait en effectuant l'opération *put(id, commande)* sur le sac *Actionneurs* de l'objet PUTUTU *Plugwise* ;
- **d'un commutateur EnOcean** [43] : il est utilisé comme un capteur de présence. Le commutateur a un bouton qui peut être pressé pour émuler une présence. La valeur du commutateur (présence détectée ou non) est obtenue en effectuant l'opération *rd(id,val)* sur le sac *Sensors* l'objet *EnOcean* ;
- **d'une interface graphique** : elle est utilisée pour émuler le volet. Un sac est créé pour envoyer une commande au volet. L'insertion d'un tuple, *(id, commande)* dans ce sac, montre l'action correspondante (ouvrir, fermer) sur l'interface ;
- **d'une Raspberry Pi** : elle est utilisée pour exécuter la règle du contrôleur et les objets de la couche d'abstraction Elle est connectée à la prise intelligente et au commutateur à travers deux dongles de type égal à *Plugwise* et *EnOcean*.

Deux sacs (*LuminositeExt* et *Agenda*) contenus dans un objet (*Piece*) ont été créés pour émuler un capteur de luminosité extérieur et un agenda pour les réunions.

Plusieurs scénarios ont été effectués pour valider le démonstrateur. Un des scénarios contient un conflit (présence détectée, luminosité extérieure entre [500, 600] lux et réunion confidentielle en cours). D'autres scénarios contiennent des erreurs de communication ou des panes matérielle (p. ex., prise débranchée). Dans tous les cas, il n'y avait pas de conflits et pas d'inconsistances. Trois exemples de scénarios sont :

- **Premier scénario** : le bouton du commutateur a été pressé pour émuler une présence, dans la pièce, et la luminosité extérieure a été fixée à 500 lux en insérant le tuple *("500")* dans le sac *LuminositeExt*. Une réunion confidentielle a été également émulée en insérant le tuple *("reunionConfidentielle")* dans le sac *Agenda*. Dans ces conditions, le contrôleur a allumé la lampe et fermé le volet. Le conflit qui consiste à ouvrir le volet pour la lumière du jour

TABLE 6.3 – Comparaison des temps de synthèse de contrôleurs

Pièces considérées	**Temps de synthèse**
*Piece*1 (6 actionneurs/65 variables)	1.4 s
*Piece*2 (12 actionneurs/101 variables)	31 s
*Piece*3 (18 actionneurs/137 variables)	797 s
*Piece*4 (24 actionneurs /173 variables)	4888 s
*Piece*5 (48 actionneurs/353 variables)	10920 s

et le fermer au même instant du fait d'une réunion confidentielle a été évité.

— **Deuxième scénario** : il a été effectué juste après le premier scénario. La présence était encore détectée, la luminosité extérieure était égale à 500 lux, le volet était fermé et la lampe allumée. Dans ce contexte, la fin de la réunion confidentielle a été émulée en supprimant le tuple *("reunionConfidentielle")* du sac *Agenda* et en insérant le nouveau tuple *("pasDeReunionConfidentielle")*. Cela a ouvert le volet et éteint la lampe pour économiser de l'énergie.

— **Troisième Scénario** : il a été effectué après le deuxième scénario. La présence était encore détectée, la lampe éteinte et le volet ouvert. La luminosité extérieure a été fixée à 700 lux. De plus, une panne a été émulée sur la lampe (la prise intelligente a été débranchée). Dans ce cas, le contrôleur décide de fermer le volet et d'allumer la lampe pour maintenir la luminosité entre 500 et 600 lux. Comme la lampe n'est pas accessible, la transaction a échoué et l'inconsistance qui consiste à supposer que la lampe a été allumée n'est pas survenue. La règle est conçue de sorte qu'elle envoie un SMS à la maintenance.

Mise en œuvre de boucles pour quatre autres pièces Quatre autres pièces (*Piece*2, *Piece*3, *Piece*4, *Piece*5) possédant les mêmes types d'actionneurs et de capteurs que la pièce *Piece*1 et qui doivent réaliser les même objectifs sont considérées. La *Piece*2 possède deux volets, deux fenêtres, deux lampes, une porte, un climatiseur réversible et une ventilation mécanique. *Piece*3 a, par rapport à la pièce *Piece*2, deux autres lampes, fenêtres et volets, et ainsi de suite jusqu'à *Piece*5.

Une boucle avec un contrôleur basé sur la synthèse de contrôleurs discrets à été mis en œuvre pour chaque pièce. Le Tableau 6.3 présente les temps de synthèse des différents contrôleurs, sur une ressource de calcul de 15 Go de RAM avec un processeur i7 (3.4 GHZ). Le Tableau illustre le fait que le temps de synthèse augmente de façon exponentielle avec le nombre de variables qui sont utilisées dans le modèle du système considéré. Ce coût peut être réduit par la synthèse de contrôleurs discrets modulaire [35]. Cela a été effectué dans [127] pour les mêmes pièces de bureau.

6.3 Conclusion

Ce chapitre a présenté l'implémentation du support intergiciel SICODAF avec des outils concrets et sa validation expérimentale à l'aide de deux études de cas.

L'implémentation a été effectuée avec l'intergiciel transactionnel LINC, le langage Heptagon/BZR, basé sur des systèmes de transitions et supportant la synthèse de contrôleurs discrets, et l'environnement d'abstraction PUTUTU. LINC a été d'abord combiné à Heptagon/BZR pour fournir les deux formes de fiabilité : fiabilité d'exécution et fiabilité comportementale. Ensuite les deux types de boucles (boucles de déploiement et boucles applicatives) ont été implémentés. Pour la boucle de déploiement, l'implémentation est relative au langage de description de systèmes, à la couche d'abstraction et au générateur de règles exécutant la fonction de transitions du contrôleur. Le langage de description a été implémenté avec l'environnement Eclipse Xtext, la couche d'abstraction avec LINC et le générateur de règles en Python. Cette implémentation est partielle car le générateur de modèle comportemental (un programme Heptagon/BZR) à partir du fichier de description d'un système et de ses objectifs n'a pas encore été développé. L'implémentation de boucles applicatives a été effectuée dans le contexte du bâtiment intelligent. Elle est relative à la couche d'abstraction, basée sur PUTUTU, et à deux générateurs de règles exécutant la fonction de transitions du contrôleur : une règle *template* et son instance.

Pour la validation de SICODAF, deux études de cas ont été présentées. La première est relative au déploiement d'un système de traitement de données et la deuxième est relative à la conception d'une pièce de bureau intelligente. Pour chacune de ces études de cas, le système considéré et ses objectifs ont été décrits. Ensuite, une boucle de déploiement (resp. une boucle applicative) a été mise en œuvre pour le système de traitement de données (resp. la pièce de bureau intelligente). Enfin, le comportement de chacune des deux boucles a été présenté. Cela a permis d'illustrer le fait que SICODAF supporte l'adaptation autonome des systèmes en garantissant à la fois leur fiabilité comportementale et leur fiabilité d'exécution.

Chapitre 7

Conclusion et Perspectives

7.1 Conclusion

Cette thèse s'est intéressée aux systèmes adaptatifs, à leur fiabilité par rapport à la prise des décisions d'adaptation et à l'exécution des actions correspondantes. Dans ce contexte, elle a proposé un support intergiciel qui permet de concevoir et de déployer des systèmes adaptatifs à l'aide de systèmes de transitions pour la fiabilité comportementale et d'un intergiciel transactionnel pour la fiabilité d'exécution.

7.1.1 Rappel du contexte et de la problématique

Les systèmes dans le contexte l'informatique pervasive et l'internet des objets sont distribués et constitués de nombreuses entités hétérogènes. De tels systèmes s'adaptent aux changements qui surviennent dans leur environnement et sont appelés systèmes adaptatifs. L'adaptation de ces systèmes consiste à collecter des données de leur environnement, les analyser pour prendre des décisions d'adaptation et exécuter les actions correspondantes dans le but de réaliser leurs objectifs et rester opérationnels. La conception de tels systèmes et leur déploiement sont rendus difficiles par plusieurs facteurs : la nature hétérogène et distribuée de ces systèmes, le risque de décisions d'adaptation conflictuelles et d'inconsistances (le fait de supposer qu'une action est effectuée alors qu'elle ne l'est pas). Les inconsistances sont causées, à l'exécution, par des erreurs de communication et des pannes matérielles.

De nombreuses solutions ont été proposées pour la conception et le déploiement de systèmes adaptatifs. Du fait de la nature distribuée de ces systèmes, plusieurs solutions utilisent un intergiciel. La plupart de ces solutions permettent d'éviter les décisions conflictuelles. Certaines solutions permettent de détecter les inconsistances et d'autres permettent de les éviter. Il y a un besoin de solutions qui permettent à la fois d'éviter les décisions conflictuelles, de détecter les inconsistances et d'empêcher, dans le cas où les inconstances peuvent être évitées, le fait qu'elles surviennent.

7.1.2 Contribution de la thèse

Cette thèse a proposé un support intergiciel, appelé SICODAF, pour la conception et le déploiement de systèmes adaptatifs fiables et autonomes. Ce support intergiciel est basé sur les principes du calcul autonomique. Il permet de concevoir et de déployer un système adaptatif sous la forme d'une boucle autonomique qui combine deux formes de fiabilité : une fiabilité comportementale (absence de décisions d'adaptation conflictuelles) et une fiabilité d'exécution (absence d'inconsistances).

Une boucle conçue à l'aide de SICODAF est constituée d'une couche d'abstraction pour l'hétérogénéité du système, d'un mécanisme d'exécution transactionnelle pour éviter les inconsistances et d'un contrôleur pour la prise de décisions d'adaptation qui sont correctes et cohérentes. Le contrôleur peut être basé sur des règles, sur le contrôle continu ou le contrôle discret. Une telle boucle peut être une boucle de déploiement ou une boucle applicative et peut être conçue de façon manuelle par le développeur, en suivant une méthodologie décrite par SICODAF. Elle peut aussi être conçue de façon semi-automatique. Dans ce cas, les composants de la boucle sont générés à partir d'un fichier de description du système et de ses objectifs.

SICODAF permet également la reconfiguration du contrôleur d'une boucle pour gérer les changements d'objectifs qui peuvent survenir dans le système. Il permet aussi d'intégrer dans une boucle, un système de détection de pannes matérielles.

Enfin, SICODAF permet la mise en œuvre de boucles multiples pour la conception et le déploiement des systèmes adaptatifs qui sont constitués d'un nombre élevé d'entités ou qui requièrent différents types de contrôleurs. Les boucles multiples peuvent être composées selon trois modes (parallèle, coordonné et hiérarchique).

SICODAF a été mis en œuvre puis validé avec trois études de cas dont deux dans le domaine du bâtiment intelligent. Ces études de cas ont montré que le support intergiciel permet l'adaptation autonome des systèmes et garantit leur fiabilité.

7.2 Perspectives

Les perspectives de cette thèse consistent, dans un premier temps, à permettre l'utilisation de SICODAF dans d'autres domaines d'application ou pour gérer d'autres problématiques. Ensuite, les perspectives consistent en l'extension de SICODAF. L'objectif est d'étendre la classe de systèmes considérés et de fournir aux développeurs plus de support pour la conception automatisée de boucles autonomiques.

7.2.1 SICODAF pour d'autres domaines ou problématiques

SICODAF peut être utilisé dans d'autres domaines d'application tels que les maisons, les usines et les villes intelligentes. Les différences entre ces domaines d'application et celui qui est considéré actuellement, le bâtiment intelligent, sont les entités à contrôler et leur modélisation. Par exemple, dans une maison intelligente, il faudrait considérer des entités telles que les machines à laver, les fours et les micro-ondes. Dans sa version actuelle, SICODAF permet aux développeurs de concevoir de façon manuelle une boucle pour un système de ces domaines, en suivant la méthodologie de conception décrite. Il serait intéressant de proposer un langage de spécification pour chacun de ces domaines. Cela permettrait à un développeur de décrire le système considéré et de générer, de façon automatique, la boucle associée. SICODAF pourrait aussi être utilisé dans d'autres domaines tels que le déploiement dans le contexte de l'informatique en nuage ou les grilles de calcul, comme dans [19].

SICODAF pourrait également être utilisé pour répondre à d'autres problématiques tels que la sécurité des systèmes. Une boucle pourrait être conçue pour pro-

téger un système contre les attaques malicieuses. Dans ce cas, la boucle collecte des données du système et les analyse pour détecter des attaques. Ensuite, elle décide de la mesure de protection à appliquer et exécute les actions correspondantes.

7.2.2 Extension de SICODAF

Les extensions qu'il serait intéressant d'apporter à SICODAF sont relatives à l'amélioration de la boucle générique et la conception automatique de boucles.

7.2.2.1 Améliorations de la boucle générique

Les améliorations pouvant être apportées à la boucle générique sont :

— **L'extension de la classe de systèmes considérés** : les systèmes considérés dans la version actuelle de SICODAF sont constitués d'une application devant être déployée sur une plateforme d'exécution. Un tel système est un cas particulier. En général, plusieurs applications avec des caractéristiques et des contraintes différentes et doivent être déployées sur la même plateforme d'exécution. Il serait intéressant de considérer cette classe de systèmes et de proposer une approche pour l'allocation des ressources à ces applications ;

— **Permettre l'adaptation prédictives des systèmes** : dans la version actuelle de SICODAF, l'adaptation des systèmes est réactive. Les données collectées correspondent aux changements qui surviennent dans l'environnement et sont utilisées pour la prise des décisions d'adaptation. Il serait intéressant de proposer une approche permettant d'anticiper les changements et d'adapter les systèmes en conséquence. Une telle approche requiert d'abord de définir pour un système, les changements qui doivent être anticipés en fonction des objectifs à réaliser. Ensuite, à définir comment l'occurrence d'un tel changement peut être prédit en fonction de l'état du système et des données ;

— **La proposition d'une approche permettant l'exécution distribuée des boucles hiérarchiques** : Dans la version actuelle de SICODAF, lorsque les boucles hiérarchiques sont conçues à l'aide de la synthèse de contrôleurs discrets modulaire, leur exécution est centralisée. Une règle invoque la fonction de transitions principale qui exécute toute les autres fonctions de transitions. Il serait intéressant de proposer une approche permettant l'exécution distribuée des contrôleurs de boucles hiérarchiques, comme illustrée dans [34].

7.2.2.2 Conception automatique de boucles

Il serait intéressant de permettre la conception automatique de boucles par

— **La proposition d'un langage pour la description des d'un bâtiment intelligent et de ses objectifs** : dans la version actuelle de SICODAF, le développeur fournit un modèle de l'environnement sous la forme d'un système de transitions. Ce modèle est ensuite utilisé pour générer le contrôleur à l'aide de la synthèse de contrôleurs discrets. La fiabilité d'un tel contrôleur dépend du modèle de l'environnement fourni par le développeur. Il serait

intéressant de proposer un outil qui permet aux développeurs de choisir, dans une liste prédéfinie, les types d'actionneurs et capteurs installés dans le bâtiment considéré et de définir les objectifs à réaliser. Cet outil devra ensuite générer le modèle de l'environnement puis le contrôleur de la boucle ;

— **La proposition d'une approche pour la conception systématique ou automatisée de contrôleurs basés sur la théorie du contrôle continu** : la conception de contrôleurs basé sur le contrôle continu pour des systèmes logiciels (p. ex., pour la répartition des charges des ressources de calcul) n'est pas une tâche facile. La raison est qu'il faut trouver un modèle de la dynamique du système. Il serait intéressant de proposer une méthode permettant d'automatiser la conception de tels contrôleurs, comme illustré dans [46] ;

— **La proposition d'une approche pour la conception automatique de boucles multiples** : il serait intéressant de proposer une approche qui à partir d'un fichier de description d'un système et de ses objectifs effectue d'abord une analyse de dépendance entre les entités du système, en termes de variables partagées ou en fonction des objectifs qui doivent être réalisés. Ensuite, elle décompose le système en sous-systèmes et génère des boucles multiples qui sont composées selon le mode de composition approprié. Le mode de composition choisi devrait minimiser le coût d'exécution du système.

Bibliographie

[1] H. Alex, M. Kumar, and B. Shirazi. Midfusion : An adaptive middleware for information fusion in sensor network applications. *Information Fusion*, 9(3) :332–343, 2008. (Cité en page 23.)

[2] R. Alur, T. A. Henzinger, F. Mang, et al. Mocha : Modularity in model checking. In *International Conference on Computer Aided Verification*, pages 521–525. Springer, 1998. (Cité en page 109.)

[3] F. Alvares, E. Rutten, and L. Seinturier. A domain-specific language for the control of self-adaptive component-based architecture. *Journal of Systems and Software*, 2017. (Cité en pages 33 et 37.)

[4] C. Andre, F. Boulanger, et al. Software implementation of synchronous programs. In *Int. Conf. on Application of Concurrency to System Design*, pages 133–142. IEEE, 2001. (Cité en pages 42 et 43.)

[5] P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23. IEEE Press, 2015. (Cité en page 32.)

[6] D. Arregui, F. Pacull, and J. Willamowski. Rule-based transactional object migration over a reflective middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 179–196. Springer, 2001. (Cité en page 41.)

[7] L. Atzori, A. Iera, and G. Morabito. The internet of things : A survey. *Computer networks*, 54(15) :2787–2805, 2010. (Cité en page 20.)

[8] J. Augusto and al. A new architecture for smart homes based on adb and temporal reasoning. In *Toward A Human-Friendly Assistive Environment : ICOST'2004, 2nd International Conference on Smart Home and Health Telematics*. IOS Press, 2004. (Cité en page 29.)

[9] J. C. Augusto and M. J. Hornos. Using simulation and verification to inform the development of intelligent environments. In *Intelligent Environments (Workshops)*, pages 413–424, 2012. (Cité en page 32.)

[10] J. C. Augusto and M. J. Hornos. Software simulation and verification to increase the reliability of intelligent environments. *Advances in Engineering Software*, 58 :18–34, 2013. (Cité en page 32.)

[11] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1) :11–33, 2004. (Cité en page 21.)

[12] C. Baier, J. P. Katoen, and k. G. Larsen. *Principles of model checking*. MIT press, 2008. (Cité en pages 28 et 32.)

[13] J. Barbosa, F. Dillenburg, G. Lermen, et al. Towards a programming model for context-aware applications. *Computer Languages, Systems & Structures*, 38(3) :199–213, 2012. (Cité en page 24.)

[14] T. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. *Software Architecture*, pages 439–480, 2005. (Cité en pages 26 et 27.)

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987. (Cité en pages 28 et 38.)

[16] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016. (Cité en page 118.)

[17] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains : modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006. (Cité en page 30.)

[18] T. Bouhadiba, Q. Sabah, G. Delaval, and E. Rutten. Synchronous control of reconfiguration in fractal component-based systems : a case study. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 309–318. ACM, 2011. (Cité en page 115.)

[19] L. Broto, D. Hagimont, P. Stolf, et al. Autonomic management policy specification in tune. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663. ACM, 2008. (Cité en page 142.)

[20] G. Cabri, L. Leonardi, and F. Zambonelli. Mars : A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4) :26–35, 2000. (Cité en page 24.)

[21] R. Calinescu, L. Grunske, M. Kwiatkowska, et al. Dynamic qos management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3) :387–409, 2011. (Cité en pages 25, 30 et 37.)

[22] J. Cano, G. Delaval, and E. Rutten. Coordination of eca rules by verification and control. In *International Conference on Coordination Languages and Models*, pages 33–48. Springer, 2014. (Cité en page 29.)

[23] V. Cardellini, E. Casalicchio, V. Grassi, et al. Moses : A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering*, 38(5) :1138–1159, 2012. (Cité en page 26.)

[24] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4) :444–458, 1989. (Cité en pages 24 et 38.)

[25] C. G. Cassandras and S. Lafortune. *Introduction to discrete event systems*. Springer Science & Business Media, 2009. (Cité en pages 15 et 57.)

[26] G. Castelli, M. Mamei, A. Rosi, and F. Zambonelli. Engineering pervasive service ecosystems : the sapere approach. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(1) :1, 2015. (Cité en page 24.)

[27] B. H. C. Cheng, R. De Lemos, H. Giese, et al. Software engineering for self-adaptive systems : A research roadmap. In *Software engineering for self-adaptive systems*, pages 1–26. Springer, 2009. (Cité en page 20.)

[28] T. Cooper. *Rule-based programming under OPS5*, volume 988. Morgan Kaufmann Publishers Inc., 1988. (Cité en page 38.)

[29] F. Corno and M. Sanaullah. Modeling and formal verification of smart environments. *Security and Communication Networks*, 7(10) :1582–1598, 2014. (Cité en page 32.)

[30] J. Coutaz, A. Demeure, S. Caffiau, and J. L. Crowley. Early lessons from the development of spok, an end-user development environment for smart homes. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing : Adjunct Publication*, pages 895–902. ACM, 2014. (Cité en page 27.)

[31] E. Curry. Adaptive and reflective middleware. *Middleware for Communications*, pages 29–52, 2004. (Cité en page 13.)

[32] P. C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In *Software Composition*, volume 4089, pages 82–97. Springer, 2006. (Cité en pages 26 et 27.)

[33] A. Dearle, G. Kirby, and A. McCarthy. A middleware framework for constraint-based deployment and autonomic management of distributed applications. *arXiv preprint arXiv :1006.4733*, 2010. (Cité en pages 25 et 26.)

[34] G. Delaval, S. M. Gueye, and E. Rutten. Distributed execution of modular discrete controllers for data center management. In *Proc. of the 5th IFAC international workshop on Dependable Control of Discrete Systems, DCDS'15*, 2015. (Cité en pages 110 et 143.)

[35] G. Delaval, S. M. Gueye, E. Rutten, and N. De Palma. Modular coordination of multiple autonomic managers. In *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*, pages 3–12. ACM, 2014. (Cité en page 138.)

[36] G. Delaval, É. Rutten, and H. Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4) :385–418, 2013. (Cité en pages 16, 37, 42 et 44.)

[37] A. Dey, T. Sohn, S. Streng, and J. Kodama. icap : Interactive prototyping of context-aware applications. *Pervasive Computing*, pages 254–271, 2006. (Cité en page 27.)

[38] J. Dubus and P. Merle. Applying omg d&c specification and eca rules for autonomous distributed component-based systems. In *MoDELS Workshops*, pages 242–251. Springer, 2006. (Cité en pages 25, 26 et 37.)

[39] L. F. Ducreux, C. Guyon-Gardeux, S. Lesecq, et al. Resource-based middleware in the context of heterogeneous building automation systems. In *IECon 2012-38th Annual Conference on IEEE Industrial Electronics Society*, pages 4847–4852. IEEE, 2012. (Cité en pages 28, 37 et 42.)

[40] E. Dumitrescu, A. Girault, H. Marchand, and É. Rutten. Multicriteria optimal reconfiguration of fault-tolerant real-time tasks. In *Workshop on Discrete Event Systems, WODES'10*, pages 366–373. IFAC, 2010. (Cité en page 33.)

[41] W. Emmerich. Software engineering and middleware : a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 117–129. ACM, 2000. (Cité en page 22.)

[42] EnOcean. Enocean. https://www.enocean.com. (Cité en pages 20, 51, 76, 90 et 92.)

[43] EnOcean. Enocean switch. https://www.enocean.com/en/enocean_modules/ptm-210/. (Cité en page 137.)

[44] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2) :114–131, 2003. (Cité en page 22.)

[45] X. Fei and E. Magill. Rule execution and event distribution middleware for prosen-wsn. In *Sensor Technologies and Applications, 2008. SENSORCOMM'08. Second International Conference on*, pages 580–585. IEEE, 2008. (Cité en page 26.)

[46] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, pages 299–310. ACM, 2014. (Cité en page 144.)

[47] C. L. Fok, G. C. Roman, and C. Lu. Agilla : A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(3) :16, 2009. (Cité en page 23.)

[48] M. García-Herranz, P. A. Haya, and X. Alamán. Towards a ubiquitous end-user programming system for smart spaces. *J. UCS*, 16(12) :1633–1649, 2010. (Cité en page 35.)

[49] D. Garlan, S. W. Cheng, A. C. Huang, et al. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, 2004. (Cité en pages 25 et 27.)

[50] K. Geihs, P. Barone, F. Eliassen, et al. A comprehensive solution for application-level adaptation. *Software : Practice and Experience*, 39(4) :385–422, 2009. (Cité en pages 26, 30 et 37.)

[51] A. Girault and É. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2) :190–225, 2009. (Cité en page 33.)

[52] G. C. Goodwin, S. F. Graebe, and M. E. Salgado. Control system design. *Upper Saddle River*, 2001. (Cité en pages 15 et 57.)

[53] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3) :843–871, 1994. (Cité en page 109.)

[54] T. Gu, H. K. Pung, and D. Q. Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and computer applications*, 28(1) :1–18, 2005. (Cité en pages 13, 26 et 37.)

[55] M. Güdemann, F. Ortmeier, and W. Reif. Formal modeling and verification of systems with self-x properties. In *ATC*, volume 4158, pages 38–47. Springer, 2006. (Cité en pages 32 et 34.)

[56] S. Guillet, B. Bouchard, and A. Bouzouane. Correct by construction security approach to design fault tolerant smart homes for disabled people. *Procedia Computer Science*, 21 :257–264, 2013. (Cité en pages 33, 34, 36 et 37.)

[57] D. Guinard, V. Trifa, S. Karnouskos, et al. Interacting with the soa-based internet of things : Discovery, query, selection, and on-demand provisioning of web services. *IEEE transactions on Services Computing*, 3(3) :223–235, 2010. (Cité en page 23.)

[58] T. Hasiotis, G. Alyfantis, V. Tsetsos, et al. Sensation : A middleware integration platform for pervasive applications in wireless sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 366–377. IEEE, 2005. (Cité en page 23.)

[59] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006. (Cité en pages 28, 32, 33 et 61.)

[60] M. U. Iftikhar and D. Weyns. A case study on formal verification of self-adaptive behaviors in a decentralized system. *arXiv preprint arXiv :1208.4635*, 2012. (Cité en page 34.)

[61] K. Jensen. Coloured petri nets. In *Petri nets : central models and their properties*, pages 248–299. Springer, 1987. (Cité en page 61.)

[62] C. Julien and G. C. Roman. Egospaces : Facilitating rapid development of context-aware mobile applications. *IEEE Transactions on Software Engineering*, 32(5) :281–298, 2006. (Cité en page 24.)

[63] J. Keeney and V. Cahill. Chisel : A policy-driven, context-aware, dynamic adaptation framework. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 3–14. IEEE, 2003. (Cité en page 26.)

[64] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003. (Cité en pages 13, 24, 25 et 51.)

[65] J. O. Kephart and R. Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1), 2007. (Cité en pages 26 et 30.)

[66] N. Khakpour, S. Jalili, C. Talcott, et al. Formal modeling of evolving self-adaptive systems. *Science of Computer Programming*, 78(1) :3–26, 2012. (Cité en pages 26, 27 et 37.)

[67] N. Khakpour, R. Khosravi, M. Sirjani, and S. Jalili. Formal analysis of policy-based self-adaptive systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2536–2543. ACM, 2010. (Cité en page 29.)

[68] D. Kolokotsa, A. Pouliezos, G. Stavrakakis, and C. Lazos. Predictive control techniques for energy and indoor environmental quality management in buildings. *Building and Environment*, 44(9) :1850–1863, 2009. (Cité en pages 26, 31, 34, 35 et 37.)

[69] C. Krupitzer, F. M. Roth, S. VanSyckel, et al. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17 :184–206, 2015. (Cité en pages 26 et 31.)

[70] M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0 : Verification of probabilistic real-time systems. In *Computer aided verification*, pages 585–591. Springer, 2011. (Cité en page 30.)

[71] I. Lanese, A. Bucchiarone, and F. Montesi. A framework for rule-based dynamic adaptation. *Trustworthly Global Computing*, pages 284–300, 2010. (Cité en pages 26 et 27.)

[72] T. Le Guilly, M. K. Nielsen, T. Pedersen, et al. User constraints for reliable user-defined smart home scenarios. *Journal of Reliable Intelligent Environments*, 2(2) :75–91, 2016. (Cité en page 29.)

[73] T. Le Guilly, J. H. Smedegård, T. Pedersen, and A. Skou. To do and not to do : constrained scenarios for safe smart house. In *Intelligent Environments (IE), 2015 International Conference on*, pages 17–24. IEEE, 2015. (Cité en page 35.)

[74] J. Lee, L. Garduño, E. Walker, and W. Burleson. A tangible programming tool for creation of context-aware applications. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 391–400. ACM, 2013. (Cité en page 27.)

[75] P. Levis and D. Culler. Maté : A tiny virtual machine for sensor networks. *ACM Sigplan Notices*, 37(10) :85–95, 2002. (Cité en page 23.)

[76] C. M. Liang, B. F. Karlsson, et al. Sift : building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 298–309. ACM, 2015. (Cité en pages 28, 29 et 37.)

[77] T. Liu and M. Martonosi. Impala : A middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 107–118, New York, NY, USA, 2003. ACM. (Cité en page 23.)

[78] Y. Liu, X. Zhang, Y. Liu, et al. Towards formal modelling and verification of pervasive computing systems. In *Transactions on Computational Collective Intelligence XVI*, pages 62–91. Springer, 2014. (Cité en page 29.)

[79] M. Louvel and F. Pacull. Linc : A compact yet powerful coordination environment. In *International Conference on Coordination Languages and Models*, pages 83–98. Springer, 2014. (Cité en pages 13, 16, 24, 26, 28, 37, 38, 42 et 119.)

[80] M. Louvel, F. Pacull, and M. I. Vergara-Gallego. Coordination scheme editor for building management systems. In *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*, pages 7052–7057. IEEE, 2016. (Cité en page 35.)

[81] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb : an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1) :122–173, 2005. (Cité en page 23.)

[82] E. Magill and J. Blum. Exploring conflicts in rule-based sensor networks. *Pervasive and Mobile Computing*, 27 :133–154, 2016. (Cité en page 29.)

[83] L. Mainetti, V. Mighali, L. Patrono, and P. Rametta. A novel rule-based semantic architecture for iot building automation systems. In *Software, Telecommunications and Computer Networks (SoftCOM), 2015 23rd International Conference on*, pages 124–131. IEEE, 2015. (Cité en pages 26 et 27.)

[84] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications : The tota approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(4) :15, 2009. (Cité en page 24.)

[85] P. J. Marrón, A. Lachenmann, D. Minder, et al. Tinycubus : a flexible and adaptive framework sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 278–289. IEEE, 2005. (Cité en page 23.)

[86] C. Maternaghan and K. J. Turner. Programming home care. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2011 5th International Conference on*, pages 485–491. IEEE, 2011. (Cité en page 27.)

[87] C. Maternaghan and K. J. Turner. Policy conflicts in home automation. *Computer Networks*, 57(12) :2429–2441, 2013. (Cité en pages 28, 29 et 35.)

[88] M. E. A. Matougui and S. Leriche. A middleware architecture for autonomic software deployment. In *ICSNC'12 : The Seventh International Conference on Systems and Networks Communications*, pages 13–20. XPS, 2012. (Cité en page 26.)

[89] D. Menasce, H. Gomaa, J. Sousa, et al. Sassy : A framework for self-architecting service-oriented systems. *IEEE software*, 28(6) :78–85, 2011. (Cité en pages 26 et 30.)

[90] modbus. Modbus. `http://www.modbus.org/`. (Cité en page 20.)

[91] O. Mokrenko, S. Lesecq, L. W., et al. Dynamic power management in a wireless sensor network using predictive control. In *Industrial Electronics Society, IECON 2014-40th Annual Conference of the IEEE*, pages 4756–4761. IEEE, 2014. (Cité en page 61.)

[92] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl. Proactive self-adaptation under uncertainty : a probabilistic model checking approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 1–12. ACM, 2015. (Cité en page 30.)

[93] S. Munir and J. A. Stankovic. Depsys : Dependency aware integration of cyber-physical systems for smart homes. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pages 127–138. IEEE, 2014. (Cité en page 35.)

[94] T. Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989. (Cité en pages 28, 32 et 61.)

[95] A. L. Murphy, G. P. Picco, and G. C. Roman. Lime : A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3) :279–328, 2006. (Cité en page 24.)

[96] A. A. Nacci, B. Balaji, P. Spoletini, et al. Buildingrules : a trigger-action based system to manage complex commercial buildings. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, pages 381–384. ACM, 2015. (Cité en pages 26, 27, 29 et 37.)

[97] M. Nakamura, K. Ikegami, and S. Matsumoto. Considering impacts and requirements for better understanding of environment interactions in home network services. *Computer Networks*, 57(12) :2442–2453, 2013. (Cité en page 35.)

[98] A. Omicini and F. Zambonelli. Tucson : a coordination model for mobile information agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, volume 138, 1998. (Cité en page 24.)

[99] P. Oreizy, M. M. Gorlick, R. N. Taylor, et al. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3) :54–62, 1999. (Cité en pages 13 et 20.)

[100] F. Pacull, L. F. Ducreux, S. Thior, et al. Self-organisation for building automation systems : Middleware linc as an integration tool. In *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*, pages 7726–7732. IEEE, 2013. (Cité en pages 16, 37 et 42.)

[101] P. Patel, S. Jardosh, S. Chaudhary, and P. Ranjan. Context aware middleware architecture for wireless sensor network. In *Services Computing, 2009. SCC'09. IEEE International Conference on*, pages 532–535. IEEE, 2009. (Cité en page 26.)

[102] T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 33–42. IEEE Press, 2012. (Cité en page 31.)

[103] T. Pedersen, T. Le Guilly, et al. A method for model checking feature interactions. In *Software Technologies (ICSOFT), 2015 10th International Joint Conference on*, volume 1, pages 1–10. IEEE, 2015. (Cité en page 32.)

[104] P. R. Pietzuch and J. M. Bacon. Hermes : A distributed event-based middleware architecture. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 611–618. IEEE, 2002. (Cité en page 23.)

[105] Plugwise. Plugwise. `https://www.plugwise.com`. (Cité en pages 20, 51 et 90.)

[106] Plugwise. Plugwise circle. `https://www.plugwise.com/circle/`. (Cité en page 137.)

[107] D. Preuveneers and Y. Berbers. Consistency in context-aware behavior : a model checking approach. In *Workshop Proceedings of the 8th International Conference on Intelligent Environments*, volume 13, pages 401–412. IOS Press, 2012. (Cité en pages 28 et 29.)

[108] D. Preuveneers and W. Joosen. Semantic analysis and verification of context-driven adaptive applications in intelligent environments. *Journal of Reliable Intelligent Environments*, 2(2) :53–73, 2016. (Cité en pages 28, 29, 36 et 37.)

[109] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. Middleware for internet of things : a survey. *IEEE Internet of Things Journal*, 3(1) :70–95, 2016. (Cité en page 22.)

[110] R. Rouvoy, P. Barone, Y. Ding, et al. Music : Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software engineering for self-adaptive systems*, pages 164–182. Springer, 2009. (Cité en pages 23 et 26.)

[111] S. M. Sadjadi and P. K. McKinley. A survey of adaptive middleware. *Michigan State University Report MSU-CSE-03-35*, 2003. (Cité en page 13.)

[112] M. Salehie and L. Tahvildari. Towards a goal-driven approach to action selection in self-adaptive software. *Software : Practice and Experience*, 42(2) :211–233, 2012. (Cité en page 26.)

[113] R. Seiger, S. Huber, P. Heisig, and U. Assmann. Enabling self-adaptive workflows for cyber-physical systems. In *International Workshop on Business Process Modeling, Development and Support*, pages 3–17. Springer, 2016. (Cité en page 29.)

[114] R. Seiger, S. Huber, and T. Schlegel. Toward an execution system for self-healing workflows in cyber-physical systems. *Software & Systems Modeling*, pages 1–22, 2016. (Cité en page 29.)

[115] M. Serna, C. J. Sreenan, and S. Fedor. A visual programming framework for wireless sensor networks in smart home applications. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2015 IEEE Tenth International Conference on*, pages 1–6. IEEE, 2015. (Cité en page 35.)

[116] C. S. Shankar, A. Ranganathan, and R. Campbell. An eca-p policy-based framework for managing ubiquitous computing environments. In *Mobile and Ubiquitous Systems : Networking and Services, 2005. MobiQuitous 2005. The*

Second Annual International Conference on, pages 33–42. IEEE, 2005. (Cité en pages 28, 29, 36 et 37.)

[117] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio. Control-theoretical software adaptation : A systematic literature review. *IEEE Transactions on Software Engineering*, 2017. (Cité en pages 31 et 32.)

[118] S. Shevtsov and D. Weyns. Keep it simplex : Satisfying multiple goals with guarantees in control-based self-adaptive systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 229–241. ACM, 2016. (Cité en pages 26, 31 et 37.)

[119] J. Singh, J. Bacon, and D. Eyers. Policy enforcement within emerging distributed, event-based systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 246–255. ACM, 2014. (Cité en pages 26 et 27.)

[120] T. Sivaharan, G. Blair, and G. Coulson. Green : A configurable and reconfigurable publish-subscribe middleware for pervasive computing. *On the Move to Meaningful Internet Systems 2005 : CoopIS, DOA, and ODBASE*, pages 732–749, 2005. (Cité en page 23.)

[121] T. Sohn and A. Dey. icap : an informal tool for interactive prototyping of context-aware applications. In *CHI'03 extended abstracts on Human factors in computing systems*, pages 974–975. ACM, 2003. (Cité en page 35.)

[122] T. G. Stavropoulos, E. S. Rigas, E. Kontopoulos, et al. A multi-agent coordination framework for smart building energy management. In *Database and Expert Systems Applications (DEXA), 2014 25th International Workshop on*, pages 126–130. IEEE, 2014. (Cité en page 27.)

[123] T. G. Stavropoulosfeca-p, E. Kontopoulos, N. Bassiliades, et al. Rule-based approaches for energy savings in an ambient intelligence environment. *Pervasive and Mobile Computing*, 19 :1–23, 2015. (Cité en page 29.)

[124] Y. Sun, X. Wang, H. Luo, and X. Li. Conflict detection scheme based on formal rule model for smart building systems. *IEEE Transactions on Human-Machine Systems*, 45(2) :215–227, 2015. (Cité en page 29.)

[125] A. N. Sylla, M. Louvel, and F. Pacull. Coordination rules generation from coloured Petri net models. In *Proceedings of the Int. Workshop on Petri Nets and Software Engineering (PNSE'15)*, pages 325–326, 2015. (Cité en page 61.)

[126] A. N. Sylla, M. Louvel, and É. Rutten. Combining transactional and behavioural reliability in adaptive middleware. In *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*, page 5. ACM, 2016. (Cité en pages 61 et 112.)

[127] A. N. Sylla, M. Louvel, E. Rutten, and G. Delaval. Design framework for reliable multiple autonomic loops in smart environments. In *Cloud and Autonomic Computing (ICCAC), 2017 International Conference on*, pages 131–142. IEEE, 2017. (Cité en page 138.)

[128] K. Terfloth, G. Wittenburg, and J. Schiller. Facts–a rule-based middleware architecture for wireless sensor networks. In *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on*, pages 1–8. IEEE, 2006. (Cité en pages 13 et 26.)

[129] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 803–812. ACM, 2014. (Cité en page 27.)

[130] D. Valério and J. da Costa. Tuning of fractional pid controllers with ziegler–nichols-type rules. *Signal Processing*, 86(10) :2771–2784, 2006. (Cité en page 32.)

[131] C. Vannucchi, M. Diamanti, G. Mazzante, et al. Symbolic verification of event–condition–action rules in intelligent environments. *Journal of Reliable Intelligent Environments*, pages 1–14, 2017. (Cité en page 29.)

[132] M. I. Vergara-Gallego, O. Mokrenko, M. Louvel, et al. Implementation of an energy management control strategy for wsns using the linc middleware. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, pages 53–58. Junction Publishing, 2016. (Cité en page 61.)

[133] P. Vromant, D. Weyns, S. Malek, and J. Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 202–207. ACM, 2011. (Cité en page 25.)

[134] E. J. Y. Wei and A. T. S. Chan. Campus : A middleware for automated context-aware adaptation decision making at run time. *Pervasive and Mobile Computing*, 9(1) :35–56, 2013. (Cité en pages 30 et 37.)

[135] D. Weyns, B. Schmerl, V. Grassi, et al. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 76–107. Springer, 2013. (Cité en page 25.)

[136] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 139–152. ACM, 2006. (Cité en page 23.)

[137] M. Zhao, G. Privat, et al. Discrete control for the internet of things and smart environments. In *Presented as part of the 8th International Workshop on Feedback Computing*, 2013. (Cité en pages 33 et 34.)

[138] Y. Zheng, A. T. S. Chan, and G. Ngai. Mcl : a mobigate coordination language for highly adaptive and reconfigurable mobile middleware. *Software : Practice and Experience*, 36(11-12) :1355–1380, 2006. (Cité en page 24.)

[139] zigbee. Zigbee. `http://www.zigbee.org`. (Cité en pages 20, 51 et 90.)

Publications liées à la thèse

— A. N. Sylla, M. Louvel, et F. Pacull. Coordination Rules Generation from Coloured Petri Net Models. In : PNSE@ Petri Nets. 2015. p. 325-326.
— A. N. Sylla, M. Louvel, F. Pacull et E. Rutten. Génération de règles de coordination à partir de réseaux de Pétri colorés. In : 10è Colloque Francophone sur la Modélisation des Systèmes Réactifs, MSR'15. 2015.
— A. N. Sylla, M. Louvel et E. Rutten. Combining transactional and behavioural reliability in adaptive middleware. In : Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware. ACM, 2016. p. 5.
— A. N. Sylla, M. Louvel, E. Rutten et G. Delaval. Design Framework for Reliable Multiple Autonomic Loops in Smart Environments. In : Cloud and Autonomic Computing (ICCAC), 2017 International Conference on. IEEE, 2017. p. 131-142.
— M. Louvel, F. Pacull, E. Rutten et A. N. Sylla. Development Tools for Rule-Based Coordination Programming in LINC. In : International Conference on Coordination Languages and Models. Springer, Cham, 2017. p. 78-96.
— A. N. Sylla, M. Louvel et E. Rutten. Design framework for reliable and environment aware management of smart environment devices. Journal of Internet Services and Applications, 2017, vol. 8, no 1, p. 16.

Résumé

Dans le contexte de l'informatique pervasive et de l'internet des objets, les systèmes sont hétérogènes, distribués et adaptatifs (p. ex., systèmes de gestion des transports, bâtiments intelligents). La conception et le déploiement de ces systèmes sont rendus difficiles par leur nature hétérogène et distribuée mais aussi le risque de décisions d'adaptation conflictuelles et d'inconsistances à l'exécution. Les inconsistances sont causées par des pannes matérielles ou des erreurs de communication. Elles surviennent lorsque des actions correspondant aux décisions d'adaptation sont supposées être effectuées alors qu'elles ne le sont pas.

Cette thèse propose un support intergiciel, appelé SICODAF, pour la conception et le déploiement de systèmes adaptatifs fiables. SICODAF combine une fiabilité comportementale (absence de décisions conflictuelles) au moyen de systèmes de transitions et une fiabilité d'exécution (absence d'inconsistances) à l'aide d'un intergiciel transactionnel. SICODAF est basé sur le calcul autonomique. Il permet de concevoir et de déployer un système adaptatif sous la forme d'une boucle autonomique qui est constituée d'une couche d'abstraction, d'un mécanisme d'exécution transactionnelle et d'un contrôleur. SICODAF supporte trois types de contrôleurs (basés sur des règles, sur la théorie du contrôle continu ou discret). Il permet également la reconfiguration d'une boucle, afin de gérer les changements d'objectifs qui surviennent dans le système considéré, et l'intégration d'un système de détection de pannes matérielles. Enfin, SICODAF permet la conception de boucles multiples pour des systèmes qui sont constitués de nombreuses entités ou qui requièrent des contrôleurs de types différents. Ces boucles peuvent être combinées en parallèle, coordonnées ou hiérarchiques.

SICODAF a été mis en œuvre à l'aide de l'intergiciel transactionnel LINC, de l'environnement d'abstraction PUTUTU et du langage Heptagon/BZR qui est basé sur des systèmes de transitions. SICODAF a été également évalué à l'aide de deux études de cas.

Abstract

In the context of pervasive computing and internet of things, systems are heterogeneous, distributed and adaptive (e.g., transport management systems, building automation). The design and the deployment of these systems are made difficult by their heterogeneous and distributed nature but also by the risk of conflicting adaptation decisions and inconsistencies at runtime. Inconsistencies are caused by hardware failures or communication errors. They occur when actions corresponding to the adaptation decisions are assumed to be performed but are not.

This thesis proposes a middleware support, called SICODAF, for the design and the deployment of reliable adaptive systems. SICODAF combines a behavioral reliability (absence of conflicting decisions) by means of transitions systems and an execution reliability (absence of inconsistencies) through a transactional middleware. SICODAF is based on autonomic computing. It allows to design and deploy an adaptive system in the form of an autonomic loop which consists of an abstraction layer, a transactional execution mechanism and a controller. SICODAF supports three types of controllers (based on rules, on continuous or discrete control theory). SICODAF also allows for loop reconfiguration, to deal with changing objectives in the considered system, and the integration of a hardware failure detection system. Finally, SICODAF allows for the design of multiple loops for systems that consist of a high number of entities or that require controllers of different types. These loops can be combined in parallel, coordinated or hierarchical.

SICODAF was implemented using the transactional middleware LINC, the abstraction environment PUTUTU and the language Heptagon/BZR that is based on transitions systems. SICODAF was also evaluated using two case studies.